Warwick Guide to J
(Version 3.05c)

J. E. H. Shaw

August 18, 1999

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 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!

Acknowledgment

I am indebted to many people for their comments on earlier drafts of this Guide, particularly Chris Burke, John Howland, Ken Iverson, Walter Johnston, David Ness, David Vincent-Jones, and <*insert your name here*>. So far I have only made minor alterations in the light of their comments; a more extensive revision should appear around June 1998.

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 **6**
2.1 Vocabulary . . . 6
2.1.1 Verbs . . . 6
2.1.2 Nouns . . . 6
2.1.3 Adverbs . . . 8
2.1.4 Conjunctions . . . 8
2.1.5 J Words in Context . . . 8
2.2 Data . . . 9
2.2.1 Characters . . . 10
2.2.2 Numbers . . . 11
2.2.3 Boxes . . . 18
2.2.4 Arrays . . . 18
2.2.5 Data Structures . . . 24
2.3 Grammar . . . 26
2.3.1 Order of Evaluation . . . 26
2.3.2 Trains and Hooks and Forks . . . 27
2.3.3 Some Adverbs . . . 31
2.3.4 Actions on Arrays . . . 34
2.3.5 Matrix Algebra . . . 39
2.3.6 More Conjunctions . . . 45
2.3.7 Miscellaneous Constructions . . . 50
2.4 Programming . . . 53
2.4.1 Assignment . . . 53
2.4.2 Tacit Definition . . . 53
2.4.3 Explicit Definition . . . 53
2.4.4 Recursion and Self-Reference . . . 53
2.4.5 Examples . . . 54
2.4.6 Control Structure . . . 56

3 J Implementation **57**
3.1 J for Windows . . . 57
3.2 Foreign Conjunction . . . 57
3.2.1 Families of Foreign Conjunctions . . . 58
3.2.2 List of Foreign Conjunctions . . . 59
3.3 Associated Scripts and Packages . . . 60
3.3.1 Mathematics . . . 60
3.3.2 Regular Expressions . . . 60
3.4 Lab Sessions . . . 60
3.5 Package Development . . . 60
3.5.1 Locales . . . 60

3.5.2 Debugging 60
3.5.3 Distribution 60
3.6 Graphics Interface 61
3.7 J Support 61
3.7.1 Help Files 61
3.7.2 J Books and Papers 61
3.7.3 J on the Net 61

4 J Examples **62**
4.1 Utilities 62
4.1.1 Standard J Utilities 62
4.1.2 My Utilities 62
4.2 Statistics 67
4.3 Timetabling 68
4.3.1 J Timetabling Script 68
4.3.2 TeX Timetables Produced by J 74

References 79

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 modifed 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 `CRLF` (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 implemention and communication: 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 including examples and help files takes around 5 Mb disk space, and an application including executables and scripts can easily fit on a diskette. 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 home page `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 soonest, 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 Vocabulary

The full J Vocabulary, extracted and augmented from the J Dictionary help file, is shown in Table 2.1. Note that `I.`, `S:`, `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

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:

```
   < 3 1 4 1       NB. Monadic verb < means 'box y.'
+-------+
|3 1 4 1|
+-------+
   3 1 < 4 1       NB. Dyadic < means 'is x. less than y.?' (1 if true, 0 if false)
1 0
```

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.

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[\]^_
```

=	Self-Classify — Equal	=.	Is (Local)	=:	Is (Global)
<	Box — Less Than	<.	Floor — Lesser of	<:	Decrem — Less Or Eq
>	Open — Larger Than	>.	Ceiling — Larger of	>:	Increm — Larg Or Eq
_	Negative Sign / *Infinit*	_.	*Indeterminate*	_:	Infinity
+	Conjugate — Plus	+.	Real/Imag — GCD (Or)	+:	Double — Not-Or
*	Signum — Times	*.	Len/Angle — LCM (And)	*:	Square — Not-And
-	Negate — Minus	-.	Not — Less	-:	Halve — Match
%	Reciprocal — Divided by	%.	Matrix Inv — Mat Divide	%:	Square Root — Root
^	Exponential — Power	^.	Natural Log — Logarithm	^:	**Power**
$	Shape Of — Shape	$.		$:	Self-Reference
~	***Reflex***—***Pass***—Evoke	~.	Nub	~:	Nub Sieve — Not-Eq
|	Magnitude — Residue	|.	Reverse — Rotate (Shift)	|:	Transpose
.	**Det** — **Dot Product**	..	**Even**	.:	**Odd**
:	**Explicit** — **Monad/Dyad**	:.	**Obverse**	::	**Adverse**
,	Ravel — Append	,.	Ravel Items — Stitch	,:	Itemize — Laminate
;	Raze — Link	;.	**Cut**	;:	Word formation
#	Tally — Copy	#.	Base 2 — Base	#:	Antibase 2 — Antib
!	Factorial — Out Of	!.	**Fit (Customize)**	!:	**Foreign**
/	***Insert***—***Table***—Insert	/.	***Oblique***—***Key***—Append	/:	Grade Up — Sort
\	***Prefix***—***Infix***—Train	\.	***Suffix***—***Outfix***	\:	Grade Down — Sort
[	Same — Left	[.	**Lev**	[:	Cap
]	Same — Right	].	**Dex**	]:	*Identity*
{	Catalogue — From	{.	Head — Take	{:	Tail
}	***Item Amend*** — ***Amend***	}.	Behead — Drop	}:	Curtail
"	**Rank** — Constant	".	Do	":	Format
‘	**Tie (Gerund)**	‘.		‘:	**Evoke Gerund**
@	**Atop**	@.	**Agenda**	@:	**At**
&	**Bond/Compose**	&.	**Under (Dual)**	&:	**Appose**
?	Roll — Deal	?.	Roll — Deal (fixed seed)	?:	
{::	Map — Fetch				
a.	*Alphabet*	a:	*Ace*	A.	Atomic Permute
b.	***Boolean/Basic***	c.	Characteristic values	C.	Cycle-Dir — Perm
D.	**Derivative**	D:	**Secant Slope**	e.	Raze In — Memb
E.	— Member of Interval	f.	***Fix***	H.	**Hypergeometric**
i.	Integer — Index of	I.	***Integral***	j.	Imaginary — Complex
L.	Level	L:	**Level**	NB.	Comment
o.	Pi Times — Circlr	p.	Polynomial	p:	Prime
q:	Prime Factors	r.	Angle — Polar	S:	**Spread**
t.	***Assign Taylor coef***	t:	***Wgtd Taylor coef***	T.	**Taylor Function**
x:	Extend	_9: to 9:	Constant functions		

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

2.1.3 Adverbs

Adverbs modify the action of verbs, typically producing a new verb. 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' or 'add up'):

```
   +/ 3 1 4 1      NB. +/ inserts + between each item of y. & evaluates result
9
```

Note that adverbs come *after* the verb they modify, as usually happens in English ('run quickly', 'run backwards'). As in English, adverbs let you greatly extend the power of the language without having to introduce a new word for each new idea.

2.1.4 Conjunctions

Conjunctions are like adverbs but take *two* arguments, typically producing a new verb. 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     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 `"` ('CONSTANT') creates a constant verb with value `x.` and *rank* (as explained in Section 2.3.4.1) `y.`:

```
   (100"_) 1       NB. 100"_ is a verb returning 100 for any argument y.
100
```

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         NB. reflexive, i.e. (3^3), (4^4), (5^5), (6^6)
27 256 3125 46656
   3 4 ^ ~ 5 6         NB. passive tense, i.e. (5^3), (6^4)
125 1296
   'mean' ~ 3 4 5 6    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'
```

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

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: `meandata` 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:

```
   'c' , 'd'
cd
   3 , 1
3 1
   'c' , 3
|domain error
|   '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
┌─┬─┬───────────────────┬────────────────────┬─────────────┐
│c│3│┌─────────────────┐│a list of characters│3 1 4 1 59.26│
│ │ ││something in a box││                    │             │
│ │ │└─────────────────┘│                    │             │
└─┴─┴───────────────────┴────────────────────┴─────────────┘
```

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:

```
   'qwertyuiop' i. 'typewriter'
4 5 9 2 1 3 7 4 2 3
   'qwertyuiop' 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
   'qwertyuiop' i. 'what if RHS omits some chars?'
1 10 10 4 10 7 10 10 10 10 10 10 8 10 7 4 10 10 10 8 10 2 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), and if the font in use has line-drawing characters, then `179{a.` draws a vertical line, and `196{a.` draws a horizontal line:

```
   218 194 196 191 10 195 197 196 180 10 179 179 65 179 10 192 193 196 217 { a.
┌┬─┐
├┼─┤
││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.

Similarly, J recognises that the result of `5.14159-3.14159` is the integer `2` rather than another floating point number. Such *type conversions* are done automatically, so you typically needn't worry about the number type in use.

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 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 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 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
1 1 1 1 1 0 1 0 1 1 1 0 0 0 0
   5 > 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9     NB. greater than
1 1 1 1 0 0 1 0 0 1 0 0 0 0 0
```

and the standard *Boolean* (or *logical*) *functions* `-.` (*not*), `+.` (*or*), `*.` (*and*), `+:` (*not or*), `*:` (*not and*):

```
   -. 0 1                NB. not
1 0
   0 1 0 1 +. 0 0 1 1    NB. logical or
0 1 1 1
   0 1 0 1 *. 0 0 1 1    NB. logical and
0 0 0 1
   0 1 0 1 +: 0 0 1 1    NB. not or
1 0 0 0
   0 1 0 1 *: 0 0 1 1    NB. not and
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    NB. exclusive or (xor)
0 1 1 0
   0 1 0 1 <: 0 0 1 1    NB. implies
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 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
   '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
```

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. This illustrates how J distinguishes clearly between function and notation:

```
   123 _123 12e5 12000e_2    NB. a list of four integers
123 _123 1200000 120
   123 -123 12e5 12000e_2    NB. 123 minus a list of three integers
0 _1199877 3
```

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 0 1 1 1
   - 123 _123 _1 0 1 199999 200000   NB. negate
_123 123 1 0 _1 _199999 _200000
   | 123 _123 _1 0 1 199999 200000   NB. magnitude (absolute value of)
123 123 1 0 1 199999 200000
   ! i. 9      NB. factorial 0 1 ... 8
1 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 1 1 1 1 0 1 1
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 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 0 0 0 0 1
1 1 0 0 0 0 1 1 0 1 0 0 1 1 1 1 1 1
1 1 0 0 0 0 1 1 0 1 0 1 0 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:

```
   16 #. 3 14 15 9          NB. base 16 interpretation of y.
16121
   16 16 16 16 #: 16121     NB. inverse of above operation
3 14 15 9
   _ 24 60 60 #: 1000000    NB. a million seconds = 11 days 13 hrs 46 mins 40 secs.
11 13 46 40
   1 #. 1 1 41    NB. polynomial evaluation: x^2 + x + 41 evaluated at x = 1
43
   2 #. 1 1 41    NB. x^2 + x + 41 evaluated at x = 2
47
   3 #. 1 1 41    NB. x^2 + x + 41 evaluated at x = 3
53
```

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':

```
   ] p=. p: 1393 + i. 11     NB. 1393rd, 1394th, ... 1404th prime
11587 11593 11597 11617 11621 11633 11657 11677 11681 11689 11699
   4|p     NB. 9 consecutive primes are of form 4n+1, starting with 11593
3 1 1 1 1 1 1 1 1 1 3
   ] y=. 199 + 210 * i. 12   NB. arithmetic progression
199 409 619 829 1039 1249 1459 1669 1879 2089 2299 2509
   (p: i. 1000) i. y         NB. indices of y in first 1000 primes
45 79 113 144 174 203 231 262 288 315 1000 1000
   NB. i.e. 199 to 2089 are prime, 2299 and 2509 are composite
   q: 2299                   NB. prime factors of 2299
11 11 19
   __ q: 2299 2509           NB. __ q: y.  returns prime factors, multiplicities
11  19
 2   1

13 193
 1   1
```

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
   (10x^30) - a   NB. 10x^30 is evaluated precisely since 30 is coerced to 30x
999999999998765432109876543211
   q: a           NB. prime factorisation of a
3 3 101 3541 3607 3803 27961
   q: >: 2x^32    NB. Fermat wrongly hypothesised that 1 + 2^32 is prime
641 6700417
   341 | 2^340x   NB. residue (341 = 11*31 is a 'pseudoprime in base 2')
1
   !50x           NB. factorial 50
30414093201713378043612608166064768844377641568960512000000000000
   __ q: !50x     NB. prime factorisation of !50
 2  3  5 7 11 13 17 19 23 29 31 37 41 43 47
47 22 12 8  4  3  2  2  2  1  1  1  1  1  1
   ] b=. 2x^100
1267650600228229401496703205376
   b +. !50x      NB. GCD of two extended integers
140737488355328
   2x^47
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
30414093201713378043612608166064768844377641568960512000000000000
```

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. 10x^50                        NB. 10^50 times pi
314159265358979323846264338327950288419716939937510
   <.@%: ,. 2 3 4 5 6 7*10x^100        NB. 10^50 times square roots
141421356237309504880168872420969807856967187537694
173205080756887729352744634150587236694280525381038
200000000000000000000000000000000000000000000000000
223606797749978969640917366873127623544061835961152
244948974278317809819728407470589139196594748065667
264575131106459059050161575363926042571025918308245
```

2.2.2.4 Rational

Rational numbers are represented as [*numerator*]r[*denominator*], and are stored with positive denominator and with relatively prime numerator and denominator. Arithmetic on rationals is carried out to arbitrary precision:

```
   1r12 _4r123 16r_1234 _64r_12345 256r123456         NB. J simplifies these
1r12 _4r123 _8r617 64r12345 4r1929
   1r20 * 2r19 * 3r18 * 4r17 * 5r16 * 6r15            NB. equivalent to % 6!20
1r38760
   ] x=. 14142135623730950488r10000000000000000000    NB. roughly sqrt(2)
1767766952966368811r1250000000000000000
   x^2
3124999999999999999925368228735537 21r1562500000000000000000000000000000000
```

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 x: y.` returns separately the numerator and denominator of the rational approximation to `y.`

```
   x: 0.25 0.2 0.333333 0.3333333 0.33333333
1r4 1r5 333333r1000000 3333333r10000000 1r3
   +/ % x: >: i. 100      NB. sum of reciprocals from 1 to 100
14466636279520351160221518043104131447711r2788815009188499086581352357412492142272
   +/ x: % >: i. 100      NB. this is also exact because of inbuilt tolerance
14466636279520351160221518043104131447711r2788815009188499086581352357412492142272
   x: 1x1 1p1 1p4         NB. rational approximations to e, pi, pi^4
6157974361033r2265392166685 1285290289249r409120605684 24135758477270r247777268231
   ] e=. +/ x: % ! i. 20  NB. sum of reciprocals of factorial 0, 1, ..., 19
82666416490601r30411275102208
   x:^:_1 e               NB. floating point representation of above
2.71828
   2 x: 1x1 1p1 1p4       NB. numerators, denoms. of approximations to e, pi, pi^4
 6157974361033 2265392166685
 1285290289249  409120605684
24135758477270  247777268231
```

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: `0.123` is a valid number but `.123` isn't (the reason being that `.`, like `-`, is a primitive in its own right). Also note that, as well as the common notation `e` or `E` meaning 'times ten to the power…', J allows `p` meaning 'times π to the power…', and `x` meaning 'times e to the power…':

```
   1e1 2e1 _100e1 1e2 1e_1     NB. 10, 2*10, _100*10, 10^2, 1/10
10 20 _1000 100 0.1
   1p1 2p1 _100p1 1p2 1p_1     NB. pi, 2pi, _100pi, pi^2, 1/pi
3.14159 6.28319 _314.159 9.8696 0.3183099
   1x1 2x1 _100x1 1x2 1x_1     NB. e, 2e, _100e, e^2, 1/e
2.71828 5.43656 _271.828 7.38906 0.3678794
   1r3p1 22r7p_1 113r355p1     NB. pi/3, (22/7)/pi, (113/355)pi
1.0472 1.0004 0.9999999
```

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

```
   ] 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 1 1 1 1 1 1
   isinteger x            NB. 1 + 1e_14 is rounded to 1, but 1 + 1e_13 isn't
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 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*):

```
   ] 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)
_124 _4.14159 _2.5 _1 0 1.71828 2.14159 9
   >: v     NB. increment v (by 1)
_122 _2.14159 _0.5 1 2 3.71828 4.14159 11
   - v      NB. negate v
123 3.14159 1.5 0 _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 0 2 5.43656 6.28319 20
   *: v     NB. square v
15129 9.8696 2.25 0 1 7.38906 9.8696 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 0.5 1.35914 1.5708 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)
3.8175e_54 0.04321392 0.2231302 1 2.71828 15.1543 23.1407 22026.5
   o. v     NB. pi times v
_386.416 _9.8696 _4.71239 0 3.14159 8.53973 9.8696 31.4159
   %: 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 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
   -: %: 1p1 4 9r4p1 16   NB. half of square root of (pi, 4, 9pi/4, 16)
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 0 1 1x1 1p1 1e1
_123 _3.14159 _1.5 0 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 _0.4774648 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 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 1
   0 <. v          NB. lesser of 0 and v
_123 _3.14159 _1.5 0 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 0.4771213 1 3 3.69897
   5 3 2 1 0.5 1r3 _1 _2 %: 100             NB. fifth root, cube root etc. of 100
2.51189 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...
2.51189 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
31415 0.9265359
   _ 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

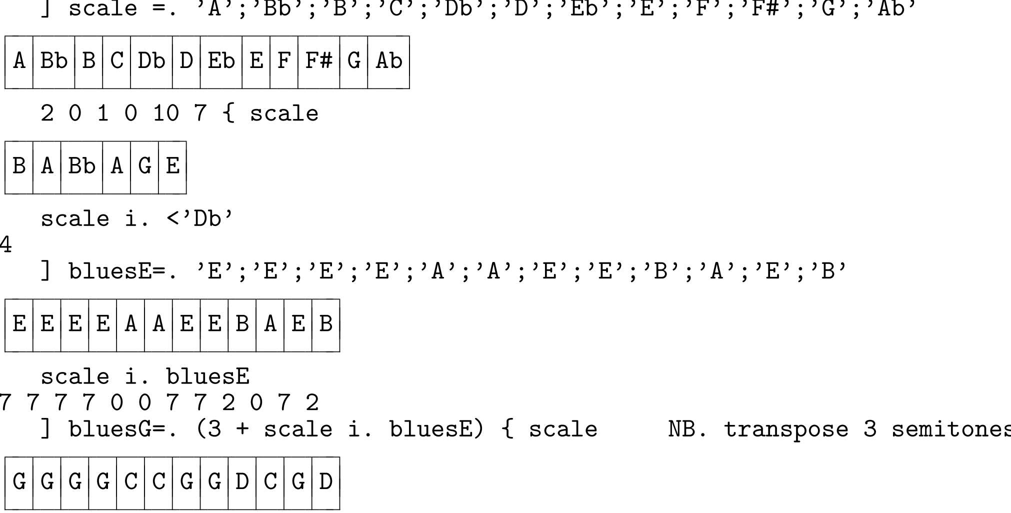
Complex numbers are represented in J by [*real part*]`j`[*imaginary part*]. For example, the following dialogue demonstrates that $\sqrt{-1} = i$, $1 + e^{i\pi} = 0$, $\log(i) = i\pi/2$, and that $-\frac{1}{2} + i\frac{\sqrt{3}}{2}$ is a complex cube root of 1:

```
   %: _1
0j1
   1 + ^0j1p1
0
   ^. 0j1
0j1.5708
   w=. _0.5 + %: _0.75
   w ^ 1 2 3
_0.5j0.8660254 _0.5j_0.8660254 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 9. 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|A|G|E|
   scale i. <'Db'
4
   ] bluesE=. 'E';'E';'E';'E';'A';'A';'E';'E';'B';'A';'E';'B'
|E|E|E|E|A|A|E|E|B|A|E|B|
   scale i. bluesE
7 7 7 7 0 0 7 7 2 0 7 2
   ] bluesG=. (3 + scale i. bluesE) { scale     NB. transpose 3 semitones
|G|G|G|G|C|C|G|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 the first *axis* and columns as the second *axis*.

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
   i. 2 3 8     NB. a 3-dimensional array
 0  1  2  3  4  5  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23

24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
   i. 0         NB. an empty list

   NB. J responded with a blank line!
```

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

```
xrs=: , (,: $ each) @: (do each)     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'
```

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

Comments:

1. Note the way that J displays arrays: `i. 4` shows a list of 4 numbers, `i. 4 3` shows 4 successive 1-dimensional arrays, `i. 2 3 4` shows 2 successive 2-dimensional arrays (separated by 1 space), `i. 2 3 1 4` shows 2 successive 3-dimensional arrays (separated by 2 spaces), etc.
2. Although the results of `i. 1` and `i. 1 1` are displayed the same way, they are in fact different, as demonstrated by their shapes (`1` and `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 *items* in the array. Thus `i. 4` has 4 items each of which is an integer; `i. 4 3` has 4 items, each being a list of 3 integers; `i. 2 3 4` has 2 items, each being a (3×4) 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:

```
s=. 1p1 [ v=. i. 3 [ m=. i. 4 2 [ A=. i. 2 3 4
xrs cut 's v m A #s #v #m #A $s $v $m $A $$s $$v $$m $$A'
```

s	v	m	A	#s	#v	#m	#A	$s	$v	$m	$A	$$s	$$v	$$m	$$A
3.14159	0 1 2	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	1	3	4	2		3	4 2	2 3 4	0	1	2	3
	3	4 2	2 3 4					0	1	2	3	1	1	1	1

Comments:

1. `[` (*left*) is a dyadic verb that returns its left argument `x.`. In the above dialogue it can be read as a statement separator.
2. `cut` is a J utility in script `strings` (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 `1p1` (π) 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:

```
v=. i.4 [ m=. i.2 4
xrs 'v';'3#v';'2 3$v';'m';'3#m';'3$m';'2 3$m'
```

v	3#v	2 3$v	m	3#m	3$m	2 3$m
0 1 2 3	0 0 0 1 1 1 2 2 2 3 3 3	0 1 2 3 0 1	0 1 2 3 4 5 6 7	0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 4 5 6 7 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
4	12	2 3	2 4	6 4	3 4	2 3 4

2.2.4.2 Reordering Axes (`|.` and `|:`)

The expression `|. y.` (*reverse* `y.`) returns `y.` with the order of its items reversed, and `x.|.y.` (`x.`*rotate* `y.`) returns `y.` with its list of items rotated leftwards `x.` times:

```
   v=. 'abcde' [ m=. i. 5 3
   xrs cut 'v m |.v 2|.v 42|.v _2|.v |.m 2|.m 42|.m _2|.m'
```

<table>
<tr><td>v</td><td>m</td><td>|.v</td><td>2|.v</td><td>42|.v</td><td>_2|.v</td><td>|.m</td><td>2|.m</td><td>42|.m</td><td>_2|.m</td></tr>
<tr><td>abcde</td><td>0 1 2
3 4 5
6 7 8
9 10 11
12 13 14</td><td>edcba</td><td>cdeab</td><td>cdeab</td><td>deabc</td><td>12 13 14
9 10 11
6 7 8
3 4 5
0 1 2</td><td>6 7 8
9 10 11
12 13 14
0 1 2
3 4 5</td><td>6 7 8
9 10 11
12 13 14
0 1 2
3 4 5</td><td>9 10 11
12 13 14
0 1 2
3 4 5
6 7 8</td></tr>
<tr><td>5</td><td>5 3</td><td>5</td><td>5</td><td>5</td><td>5</td><td>5 3</td><td>5 3</td><td>5 3</td><td>5 3</td></tr>
</table>

Monadic `|:` (*transpose*) reverses the order of the axes of `y.` (matrix transpose, where `y.` has rank 2, is just a special case). Dyadic `|:` reorders the axes of `y.` so that the `x.`th axes come last, as illustrated below:

```
   xrs 'i234'; '|:i234'; '0|:i234'; '1|:i234'; '0 1|:i234'; '0 2|:i234'
```

<table>
<tr><td>i234</td><td>|:i234</td><td>0|:i234</td><td>1|:i234</td><td>0 1|:i234</td><td>0 2|:i234</td></tr>
<tr><td>0 1 2 3
4 5 6 7
8 9 10 11

12 13 14 15
16 17 18 19
20 21 22 23</td><td>0 12
4 16
8 20

1 13
5 17
9 21

2 14
6 18
10 22

3 15
7 19
11 23</td><td>0 12
1 13
2 14
3 15

4 16
5 17
6 18
7 19

8 20
9 21
10 22
11 23</td><td>0 4 8
1 5 9
2 6 10
3 7 11

12 16 20
13 17 21
14 18 22
15 19 23</td><td>0 4 8
12 16 20

1 5 9
13 17 21

2 6 10
14 18 22

3 7 11
15 19 23</td><td>0 1 2 3
12 13 14 15

4 5 6 7
16 17 18 19

8 9 10 11
20 21 22 23</td></tr>
<tr><td>2 3 4</td><td>4 3 2</td><td>3 4 2</td><td>2 4 3</td><td>4 2 3</td><td>3 2 4</td></tr>
</table>

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

```
   xrs 'i234'; '(<0 1)|:i234'; '(<0 2)|:i234'; '(<1 2)|:i234'; '(<0 1 2)|:i234'
```

<table>
<tr><td>i234</td><td>(<0 1)|:i234</td><td>(<0 2)|:i234</td><td>(<1 2)|:i234</td><td>(<0 1 2)|:i234</td></tr>
<tr><td>0 1 2 3
4 5 6 7
8 9 10 11

12 13 14 15
16 17 18 19
20 21 22 23</td><td>0 16
1 17
2 18
3 19</td><td>0 13
4 17
8 21</td><td>0 5 10
12 17 22</td><td>0 17</td></tr>
<tr><td>2 3 4</td><td>4 2</td><td>3 2</td><td>2 3</td><td>2</td></tr>
</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│abcdmnop│abcd │
│ 4  5  6  7│efgh│qrst│┌────┬────┐│efgh│efghqrst│efgh │
│ 8  9 10 11│ijkl│uvwx││abcd│mnop││ijkl│ijkluvwx│ijkl │
│           │    │    ││efgh│qrst││mnop│        │     │
│12 13 14 15│    │    ││ijkl│uvwx││qrst│        │mnop │
│16 17 18 19│    │    │└────┴────┘│uvwx│        │qrst │
│20 21 22 23│    │    │           │    │        │uvwx │
├───────────┼────┼────┼───────────┼────┼────────┼─────┤
│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 $ 'abcdefgh'
   b=. 2 2 $ 1 ; (i.2 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│ab     │abcdefgh│┌─────┬───┐│┌─┬───┬─────┬─┐  │┌─────┬───┐│1 1 1│
│cd   │        │efgh│cd     │        ││1    │0 1│││1│0 1│0 1 2│4│  ││1    │0 1││0 1 0│
│     │        │    │       │        ││     │2 3│││ │2 3│     │ │  ││     │2 3││2 3 0│
│ef   │        │    │ef     │        │├─────┼───┤│└─┴───┴─────┴─┘  │├─────┼───┤│0 1 2│
│gh   │        │    │gh     │        ││0 1 2│4  ││                 ││0 1 2│4  ││4 4 4│
│     │        │    │       │        │└─────┴───┘│                 │└─────┴───┘│     │
├─────┼────────┼────┼───────┼────────┼───────────┼─────────────────┼───────────┼─────┤
│2 2 2│8       │2 4 │1 2 2 2│8       │2 2        │4                │1 2 2      │5 3  │
└─────┴────────┴────┴───────┴────────┴───────────┴─────────────────┴───────────┴─────┘
```

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:

```
   a=. i. 8 2
   xrs (cut 'a {.a {:a }.a }:a 3{.a _3{.a 3}.a _3}.a'),'3 1{.a';'_3 1}.a'
┌─────┬───┬─────┬─────┬─────┬────┬─────┬─────┬─────┬──────┬───────┐
│a    │{.a│{:a  │}.a  │}:a  │3{.a│_3{.a│3}.a │_3}.a│3 1{.a│_3 1}.a│
├─────┼───┼─────┼─────┼─────┼────┼─────┼─────┼─────┼──────┼───────┤
│ 0  1│0 1│14 15│ 2  3│ 0  1│0 1 │10 11│ 6  7│0 1  │0     │1      │
│ 2  3│   │     │ 4  5│ 2  3│2 3 │12 13│ 8  9│2 3  │2     │3      │
│ 4  5│   │     │ 6  7│ 4  5│4 5 │14 15│10 11│4 5  │4     │5      │
│ 6  7│   │     │ 8  9│ 6  7│    │     │12 13│6 7  │      │7      │
│ 8  9│   │     │10 11│ 8  9│    │     │14 15│8 9  │      │9      │
│10 11│   │     │12 13│10 11│    │     │     │     │      │       │
│12 13│   │     │14 15│12 13│    │     │     │     │      │       │
│14 15│   │     │     │     │    │     │     │     │      │       │
├─────┼───┼─────┼─────┼─────┼────┼─────┼─────┼─────┼──────┼───────┤
│8 2  │2  │2    │7 2  │7 2  │3 2 │3 2  │5 2  │5 2  │3 1   │5 1    │
└─────┴───┴─────┴─────┴─────┴────┴─────┴─────┴─────┴──────┴───────┘
```

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).

```
   xrs ’c’; ’1{c’; ’2 0{"1 c’; ’_1{"2 c’; ’(<1;0 2){c’; ’(<1;0 2;1 2){c’
```

c	1{c	2 0{"1 c	_1{"2 c	(<1;0 2){c	(<1;0 2;1 2){c
abcd efgh ijkl mnop qrst uvwx	mnop qrst uvwx	ca ge ki om sq wu	ijkl uvwx	mnop uvwx	no vw
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 modifed 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:

```
   a=. ’-’ ($~ $) c      NB. ’-’ (reshaped by the shape of) c
   xrs ’a’; ’’’x’’ 1}a’; ’’’xy’’ 2 0}"1 a’; ’’’x’’ _1}"2 a’; ’’’x’’ (<1;0 2)}a’
```

a	’x’ 1}a	’xy’ 2 0}"1 a	’x’ _1}"2 a	’x’ (<1;0 2)}a
---- ---- ---- ---- ---- ----	---- ---- ---- xxxx xxxx xxxx	y-x- y-x- y-x- y-x- y-x- y-x-	---- ---- xxxx ---- ---- xxxx	---- ---- ---- xxxx ---- xxxx
2 3 4	2 3 4	2 3 4	2 3 4	2 3 4

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

```
   a=. 4 10 $ ’-’
   a
----------
----------
----------
----------
   a=. ’ABCD’ ((0;2);(1;5);(3;6);<1;_1) } a
   a
--A-------
-----B---D
----------
------C---
```

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?'''
```

<table>
<tr><td>5{.42</td><td>5{.'foo'</td><td>5{.<'foo'</td><td>boxes</td><td>>boxes</td><td>>;:'eh what?'</td></tr>
<tr><td>42 0 0 0 0</td><td>foo</td><td><table><tr><td>foo</td><td></td><td></td><td></td><td></td></tr></table></td><td><table><tr><td>1</td><td>2 3 4</td></tr></table></td><td>1 0 0
2 3 4</td><td>eh
what
?</td></tr>
<tr><td>5</td><td>5</td><td>5</td><td>2</td><td>2 3</td><td>3 4</td></tr>
</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:

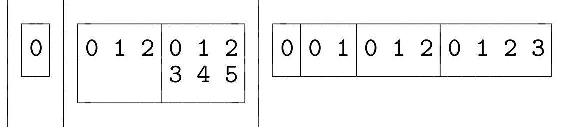
<table>
<tr><td>Monster Movie</td><td><table>
<tr><td>Father cannot yell</td><td>7 1</td></tr>
<tr><td>Mary, Mary so contrary</td><td>6 16</td></tr>
<tr><td>Outside my door</td><td>4 6</td></tr>
<tr><td>Yoo Doo Right</td><td>20 14</td></tr>
</table></td></tr>
</table>

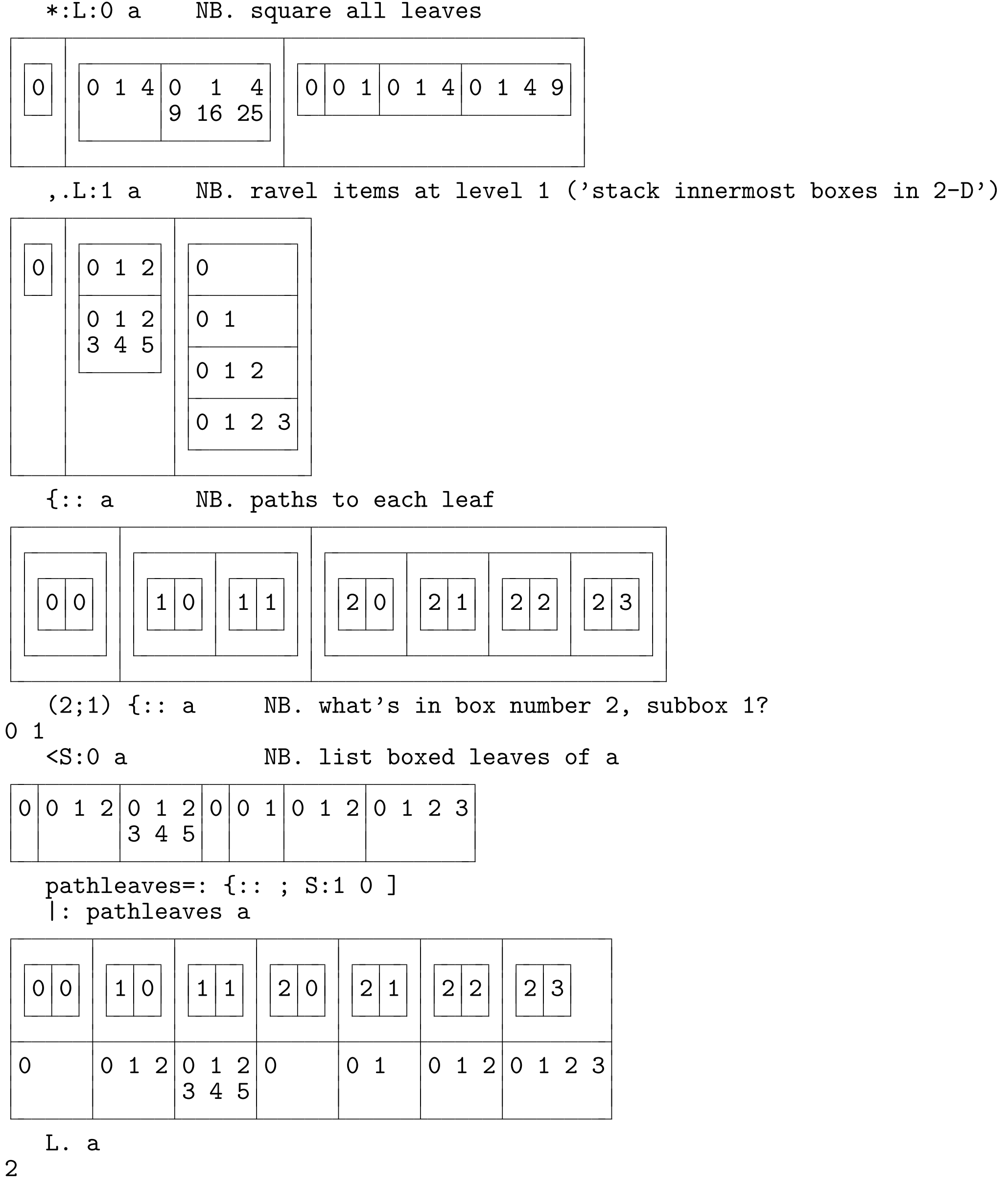
1. The first box contains the list of characters `Monster Movie`.
2. The second box contains a 4×2 array of boxes. Each of the four boxes in the first column contains a list of characters (not necessarily the same length each time). Each of the four boxes in the second column contains a list of two numbers.

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:

```
] a=. (<\ @ i.)each </. >: i. 2 2    NB. arbitrary but cute data structure
```





```
   *:L:0 a    NB. square all leaves
   ,.L:1 a    NB. ravel items at level 1 ('stack innermost boxes in 2-D')
   {:: a      NB. paths to each leaf
   (2;1) {:: a     NB. what's in box number 2, subbox 1?
0 1
   <S:0 a          NB. list boxed leaves of a
   pathleaves=: {:: ; S:1 0 ]
   |: pathleaves a
   L. a
2
```

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 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, `1x1 - +/ % ! i. n` is evaluated in the order shown below:

```
   i. 5           NB. first 5 non-negative integers
0 1 2 3 4
   ! i. 5         NB. factorials of previous result
1 1 2 6 24
   % ! i. 5       NB. reciprocals of previous result
1 1 0.5 0.1666667 0.04166667
   +/ % ! i. 5    NB. sum of previous result
2.70833
   1x1 - +/ % ! i. 5           NB. e - previous result
0.009948495
   1x1 - (+/ (% (! (i. 5))))   NB. order of evaluation made explicit
0.009948495
```

i.e. `1x1 - +/ % ! 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:

```
   -/ 1 2 3 4       NB. equivalent to 1-2-3-4
_2
   1 - 2 - 3 - 4    NB. 1 - (2 - (3 - 4))
_2
   ((1 - 2) - 3) - 4
_8
   (1 - 2) - (3 - 4)
0
   (1 - 2) + (3 - 4)
_2
   y=. 1.7 0.3 5.4 8.6 2.3
   +/ % # y         NB. sum the reciprocal of the number of items in y
0.2
   (+/ % #) y       NB. create the verb 'sum divided by number of', and apply to y
3.66
```

Note in particular that `[` (*left*) is a verb, so if you use it as a statement separator, as on page 20, 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 V_1, V_2, V_3 etc.)

2.3.2.1 Monadic Hooks

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

This interpretion 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 monadic examples of hooks are given below.

```
   lt4=. < 4:        NB. 'less than 4' (1 if true, 0 if false)
   lt4 3 1 4 1 5 9 _2 _6 _5 _3 _5
1 1 0 1 0 0 1 1 1 1 1
   isint=. = <.      NB. 1 if y. equals floor of y. (i.e. if y. is an integer)
   isint _123 _1p1 _1.5 0 1 1x1 1p1 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
   pr=. + %          NB. 'plus reciprocal'
   pr 1 2 3 4 5 100 0.1 0.01 100
2 2.5 3.33333 4.25 5.2 100.01 10.1 100.01 100.01
   issym=. -: |:     NB. 'match transpose'  i.e. is y. symmetric?
   a=. b + |: b=. *: i. 3 3 [ c=. (d=. =/~ i.4),0 [ e=. (| -/~ i.5) { 4 1 0 0 0
   xrs , (; 'issym '&,)"0 'abcde'
+---------+-------+--------+-------+-------+-------+-------+-------+---------+-------+
|a        |issym a|b       |issym b|c      |issym c|d      |issym d|e        |issym e|
+---------+-------+--------+-------+-------+-------+-------+-------+---------+-------+
| 0 10  40|1      | 0  1  4|0      |1 0 0 0|0      |1 0 0 0|1      |4 1 0 0 0|1      |
|10 32  74|       | 9 16 25|       |0 1 0 0|       |0 1 0 0|       |1 4 1 0 0|       |
|40 74 128|       |36 49 64|       |0 0 1 0|       |0 0 1 0|       |0 1 4 1 0|       |
|         |       |        |       |0 0 0 1|       |0 0 0 1|       |0 0 1 4 1|       |
|         |       |        |       |0 0 0 0|       |       |       |0 0 0 1 4|       |
+---------+-------+--------+-------+-------+-------+-------+-------+---------+-------+
|3 3      |       |3 3     |       |5 4    |       |4 4    |       |5 5      |       |
+---------+-------+--------+-------+-------+-------+-------+-------+---------+-------+
   new=. i. ~.       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 *constant functions*. J defines the 19 verbs `_9:` `_8:` ... `9:`, together with the verb `_:` (*infinity*). These each return the specified number, irrespective of the shape and value of the argument(s).
2. Dyadic `-:` (*match*) returns 1 if and only if its left and right arguments are identical (Mnemonic: `-:` resembles $\equiv$. Well, it does a bit.)
3. Monadic `~.` (*nub*) returns the distinct items of `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.

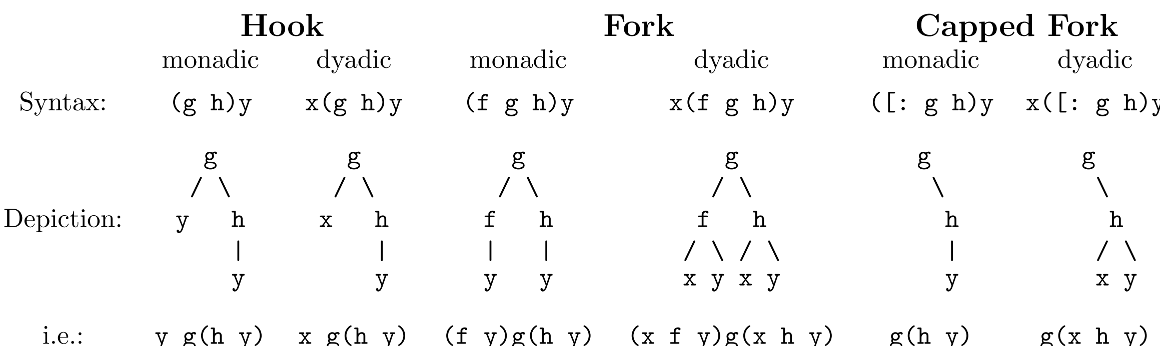


Figure 2.1: Hooks and Forks

2.3.2.2 Dyadic Hooks

The expression `x(g h)y` means `x g(h y)`, see Figure 2.1. For example, `x(-|)y` means 'x minus the absolute value of y', and the hook `- |` can be read simply as 'minus the absolute value of'.

The following dyadic examples of 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
┌──────┬────────┬──────┬───────┬─────┐
│second│smallest│fourth│largest│third│
└──────┴────────┴──────┴───────┴─────┘
   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 5.9 8.4 _9.3 9.7
   (i.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)
   dates=. 7 3 $ 4 6 1950 16 8 1944 2 3 1942 2 4 1939 9 9 1941 28 1 1945 9 2 1940
   names=. 7 7$'Dagmar Kevin  Lewis  Marvin Otis   Robert William'
   xrs 'dates'; 'names'; 'brep ''sortrev'''; 'sortrev dates'; 'names sortrev dates'
┌──────────┬───────┬──────────────┬─────────────┬───────────────────┐
│dates     │names  │brep 'sortrev'│sortrev dates│names sortrev dates│
├──────────┼───────┼──────────────┼─────────────┼───────────────────┤
│ 4 6 1950 │Dagmar │┌──┬────────┐ │ 4 6 1950    │Dagmar             │
│16 8 1944 │Kevin  ││\:│┌──┬─┬─┐│ │28 1 1945    │Robert             │
│ 2 3 1942 │Lewis  ││  ││|.│"│1││ │16 8 1944    │Kevin              │
│ 2 4 1939 │Marvin ││  │└──┴─┴─┘│ │ 2 3 1942    │Lewis              │
│ 9 9 1941 │Otis   │└──┴────────┘ │ 9 9 1941    │Otis               │
│28 1 1945 │Robert │              │ 9 2 1940    │William            │
│ 9 2 1940 │William│              │ 2 4 1939    │Marvin             │
├──────────┼───────┼──────────────┼─────────────┼───────────────────┤
│7 3       │7 7    │2             │7 3          │7 7                │
└──────────┴───────┴──────────────┴─────────────┴───────────────────┘
```

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` in script `myutil` 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 *monadic 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 0 1 0 0 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
   gmean=. # %: */      NB. 'no. of items'th root of 'product over', i.e. geometric mean
   gmean 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. (max - min) = range
   range 1.7 0.3 5.4 8.6 2.3
8.3
   eqnext=. }. = }:     NB. 1 iff current value = next value in list
   eqnext 1 3 1 1 2 2 2 1
0 0 1 0 1 1 0
   (~. ; ~:) 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9     NB. display nub & nub sieve
+-----------------+-----------------------------+
|3 1 4 5 9 2 6 8 7|1 1 1 0 1 1 1 1 0 0 0 1 0 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) * (1.0 - 0.6)
0.64
   divides=. | = 0:        NB. 1 iff x. divides y.
   12 divides 12 48 49 _132 1728 11999
1 1 0 1 1 0
   symdiff=. -. , -.~      NB. symmetric difference (items in x. but not y., or vice versa)
   3 1 4 1 5 9 2 6 5 3 5 symdiff 2 7 1 8 2 8 4 5 9 0 4 5 2
3 6 3 7 8 8 0
   'odd mark on the doormat' symdiff 'no drink or corks'
mathematics
   sdsieve=. , e. symdiff   NB. flag items of (x.,y.) that form symdiff
   3 1 4 1 5 9 2 6 5 3 5 (, ,: sdsieve) 2 7 1 8 2 8 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 1 8 2 8 4 5 9 0 4 5 2
1 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0
   _1 (, ; ,.) i. 3 4      NB. display different effects of , and ,.
+-----------+-------------+
|_1 _1 _1 _1|_1 0 1  2  3 |
| 0  1  2  3|_1 4 5  6  7 |
| 4  5  6  7|_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 (V_1V_2) where V_1 is `;` and V_2 is a fork $(V_3V_4V_5)$ in which V_5 in turn is a fork `(~: ; =)`:

```
   (; ~. ; ~: ; =) 'Canaan Banana'
┌─────────────┬─────┬───────────────────────────┬───────────────────────────┐
│Canaan Banana│Can B│1 1 1 0 0 0 1 1 0 0 0 0 0│1 0 0 0 0 0 0 0 0 0 0 0 0│
│             │     │                           │0 1 0 1 1 0 0 0 1 0 1 0 1│
│             │     │                           │0 0 1 0 0 1 0 0 0 1 0 1 0│
│             │     │                           │0 0 0 0 0 0 1 0 0 0 0 0 0│
│             │     │                           │0 0 0 0 0 0 0 1 0 0 0 0 0│
└─────────────┴─────┴───────────────────────────┴───────────────────────────┘
   dev=. - +/ % #                NB. deviations from mean
   dev 1.7 0.3 5.4 8.6 2.3
_1.96 _3.36 1.74 4.94 _1.36
   score=. +/ - >./ + <./        NB. sum excluding largest and smallest values
   score 5.9 5.9 6.0 5.8 5.6 5.9 6.0 6.0 5.9
41.4
   isposint=. (> 0:) *. <. = >.  NB. is (each atom of y.) a positive integer?
   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
1 0 0 0 0 0 0 1 0 1 0 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 ith element of the nub of its argument `y.` equals the jth element of `y.`, otherwise $m_{ij} =$ `0`.

2.3.2.5 Cap ([:)

A fork can be *capped*: `([: 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:

```
   logit=: ^. @ (% -.)       NB. logit(p) = log(p/(1-p))
   logit 0.1 0.5 0.9 0.99
_2.19722 0 2.19722 4.59512
   logit2=: [: ^. ] % -.     NB. using cap and forks to create an unbroken train
   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 `fib`, giving the `y.`th Fibonacci numbers, is utterly incomprehensible:

```
   fib=: ] ([: <. ] %~ [: >: [ ^~ [: -: [: >: ]) [: %: 5:
   fib i. 20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
   fib 30
832040
```

Even if you know that the nth Fibonacci number F_n is given by

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}, \tag{2.1}$$

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:

```
   f1=. [: -: [: >: ]    NB. f1  =  half of (1 + y.)
   f2=. [: >: f1 ^ [     NB. f2  =  1 + f1^x.
   f3=. [: <. f2 % ]     NB. f3  =  integer part of (f2 divided by y.)
   fib=. ] f3 [: %: 5:   NB. fib arg  =  f3 with (x.=arg) and (y.=sqrt(5))
   fib i. 20
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 f y. 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:

```
   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.

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

,/i.2 3 4	,./i.2 3 4	;/i.2 3 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	0 1 2 3 12 13 14 15 4 5 6 7 16 17 18 19 8 9 10 11 20 21 22 23	[0 1 2 3 4 5 6 7 8 9 10 11] [12 13 14 15 16 17 18 19 20 21 22 23]
6 4	3 8	2

Many mathematical objects can be defined naturally using insertion. For example, a *continued fraction* is an expression of the form

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cdots}}}. \tag{2.2}$$

In J this is just a1 +% a2 +% a3 +% a4 +..., or, rather prettily, (+%)/ a1,a2,a3,a4,...:

```
   (+%)/ 1 1 2 1 1 4 1 1 6 1 1 8 1 1 10        NB. CF approximation to (e-1)
1.71828
   (+%)/ 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2          NB. CF approximation to sqrt(2)
1.41421
   fibx=: {: @ (2&x:) @ ((+%)/ @ x: @ #&1)"0   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 8 9 9 9 9 9
   cumsum=. +/\             NB. cumulative sum
   cumsum i. 20             NB. triangular numbers
0 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190
   cumprod=. */\            NB. product so far
   cumprod 1x + i. 12       NB. factorials
1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600
   (+%)/\ 1 , 10#2          NB. successive continued fraction approxns. to %: 2
1 1.5 1.4 1.41667 1.41379 1.41429 1.4142 1.41422 1.41421 1.41421 1.41421
   (+%)/\ 1x , 10#2         NB. successive optimal rational approxns. to %: 2
1 3r2 7r5 17r12 41r29 99r70 239r169 577r408 1393r985 3363r2378 8119r5741
   <\ 'spoilage'            NB. box substrings
```

s	sp	spo	spoi	spoil	spoila	spoilag	spoilage

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

s	ps	ops	iops	ilops	ailops	agilops	aegilops

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. 1 only for first 1 in y.
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
   |/\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1     NB. as for </\
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
   #:/\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1    NB. as for </\
0 0 0 1 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. 0 after and including first 0 in y.
1 1 1 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 <./\
1 1 1 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 <./\
1 1 1 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. 0 only for first 0 in y.
1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
   !/\ 1 1 1 0 0 1 0 1 1 0 1 0 0 0 0     NB. as for <:/\
1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
   >./\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1    NB. 1 after and including first 1 in y.
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
   +./\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1    NB. as for >./\
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
   [/\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1     NB. fill with first item of y.
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
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
   =/\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1     NB. 1 if number of 0's so far is even
0 1 0 0 0 1 1 0 1 1 0 0 0 0 0
   -:/\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1    NB. as for =/\
0 1 0 0 0 1 1 0 1 1 0 0 0 0 0
   ~:/\ 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1    NB. 1 if number of 1's so far is odd
0 0 0 1 0 0 1 1 1 0 0 1 0 1 0
   ishead=. = {.                         NB. does item match first item?
   indelims=. [: (~:/\ *. -.) ishead     NB. 1 for items of y. in delimiters
   extract =. indelims # ]               NB. extract items delimited by {. y.
   extract '#get text#omit# delimited by #rubbish#first item# OMIT!!!'
get text delimited by first item
```

Monadic `\.` (*suffix*) similarly applies its verb right argument to each of the suffices of `y`.

```
   <\. 'zyxomma'
```

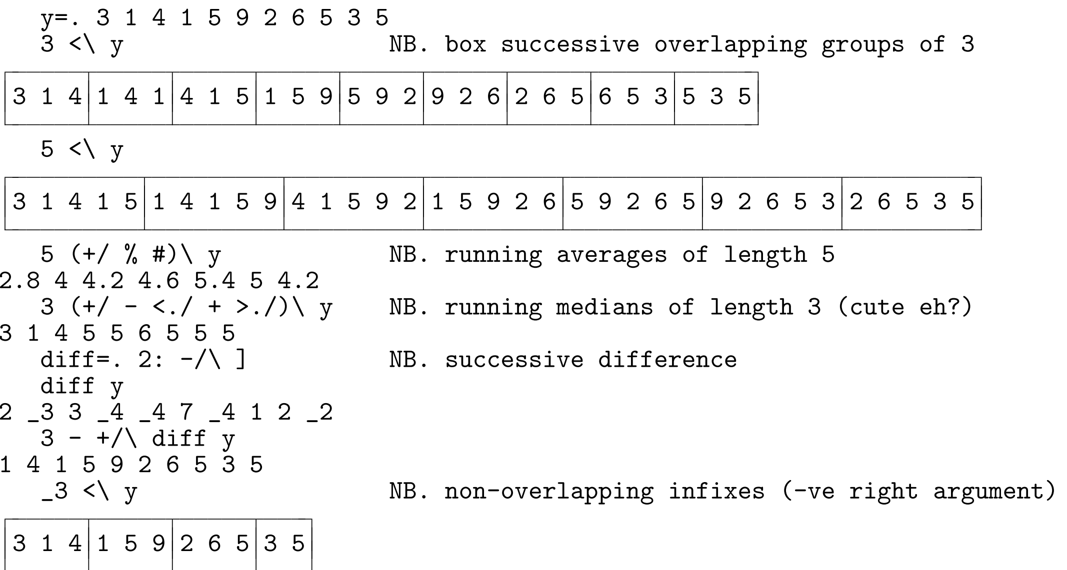
zyxomma	yxomma	xomma	omma	mma	ma	a

```
   OverRest=. /\.                   NB. example of a user-defined adverb
   <. 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 3 5 7 7 7 9
   + OverRest 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9
77 74 73 69 68 63 54 52 46 41 38 33 25 16 9
   rtb=. ([: >./\. (~:&' ')) # ]    NB. remove trailing blanks
   xrs 'rtb ''OK'''; 'rtb ''OK   '''; 'rtb '' only  TRAILING  blanks removed    '''
```

rtb 'OK'	rtb 'OK '	rtb ' only TRAILING blanks removed '
OK	OK	only TRAILING blanks removed
2	2	31

2.3.3.3 Infix (Dyadic u.\) and Outfix (Dyadic u.\.)

Dyadic `\` (*infix*) acts on *infixes* (sets of successive items of `y.`). If `x.` is positive, then `x. f\ y.` applies `f` to each infix of length `x.` in `y.`. If `x.` is negative, then `x. 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	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 4.2
   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 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
```

ProbTab	%: ProbTab	- ^. ProbTab
0.181 0.111 0.078 0.054 0.054 0.072 0.024 0.024 0.032 0.041 0.111 0.218	0.4254409 0.3331666 0.2792848 0.232379 0.232379 0.2683282 0.1549193 0.1549193 0.1788854 0.2024846 0.3331666 0.4669047	1.70926 2.19823 2.55105 2.91877 2.91877 2.63109 3.7297 3.7297 3.44202 3.19418 2.19823 1.52326
2 2 3	2 2 3	2 2 3

Similarly, the left argument of `{` (*from*) and the right argument of dyadic `i.` (*index of*) can have any shape:

```
   k=. 5 5 $ (, |.&}:) 7 1 2 6 4 1 4 3 0 6 2 3 5
   xrs 'k'; '] a=. k { ''PATERNOSTER'''; 'a. i. a'; '''PATERNOSTER'' i. a'
```

k	] a=. k { 'PATERNOSTER'	a. i. a	'PATERNOSTER' i. a
7 1 2 6 4 1 4 3 0 6 2 3 5 3 2 6 0 3 4 1 4 6 2 1 7	SATOR AREPO TENET OPERA ROTAS	83 65 84 79 82 65 82 69 80 79 84 69 78 69 84 79 80 69 82 65 82 79 84 65 83	7 1 2 6 4 1 4 3 0 6 2 3 5 3 2 6 0 3 4 1 4 6 2 1 7
5 5	5 5	5 5	5 5

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

```
   kings=. 6 7$'Edward Henry  John   RichardStephenWilliam'
   k=. 5 5 1 4 1 3 2 ,: 1 0 0 0 3 1 1
   colours=. (cut 'black blue green cyan red magenta yellow white'),. ;/2 2 2#:i. 8
   c=. 2 1 3 $ 4 7 1 7 0 7
   xrs 'kings' ; 'k' ; 'k{kings' ; 'colours' ; 'c' ; 'c{colours'
```

kings	k	k{kings	colours	c	c{colours
Edward Henry John Richard Stephen William	5 5 1 4 1 3 2 1 0 0 0 3 1 1	William William Henry Stephen Henry Richard John Henry Edward Edward Edward Richard Henry Henry	black 0 0 0 blue 0 0 1 green 0 1 0 cyan 0 1 1 red 1 0 0 magenta 1 0 1 yellow 1 1 0 white 1 1 1	4 7 1 7 0 7	red 1 0 0 white 1 1 1 blue 0 0 1 white 1 1 1 black 0 0 0 white 1 1 1
6 7	2 7	2 7 7	8 2	2 1 3	2 1 3 2

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.

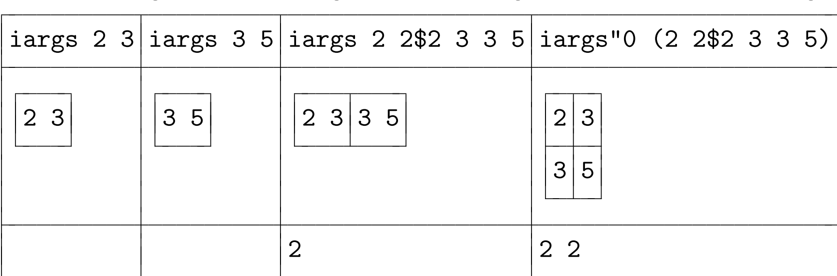
The *rank conjunction* `"` is used to specify the rank of the derived verb: V`"`k applies the verb V to the k-cells of its argument `y.`. Thus `i."0` creates integer lists whose lengths are given by the corresponding atoms of `y.`, and inserts them (padded if necessary) into a frame of shape `$y.`:

```
xrs 'i.2 3' ; 'i.3 5' ; 'i.2 2$2 3 3 5' ; 'i."0 (2 2$2 3 3 5)'
```

i.2 3	i.3 5	i.2 2$2 3 3 5	i."0 (2 2$2 3 3 5)
0 1 2 3 4 5	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14	0 1 2 0 0 3 4 5 0 0 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14	0 1 0 0 0 0 1 2 0 0 0 1 2 0 0 0 1 2 3 4
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:

```
iargs=. <"1    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)'
```



iargs 2 3	iargs 3 5	iargs 2 2$2 3 3 5	iargs"0 (2 2$2 3 3 5)
2 3	3 5	2 3 | 3 5	2 | 3 3 | 5
		2	2 2

Some other features of the intrinsic ranks of monadic verbs are shown 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 34).
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=. 3 2 2$ 1 0 0 1 3 1 5 5 3 4 2 6
xrs 'A' ; '%:A' ; '#. A' ; '%.A' ; '%."1 A' ; '$A' ; '$"2 A' ; '$"1 A'
```

A	%:A	#. A	%.A	%."1 A	$A	$"2 A	$"1 A
1 0 0 1 3 1 5 5 3 4 2 6	1 0 0 1 1.73205 1 2.23607 2.23607 1.73205 2 1.41421 2.44949	2 1 7 15 10 10	1 0 0 1 0.5 _0.1 _0.5 0.3 0.6 _0.4 _0.2 0.3	1 0 0 1 0.3 0.1 0.1 0.1 0.12 0.16 0.05 0.15	3 2 2	2 2 2 2 2 2	2 2 2 2 2 2
3 2 2	3 2 2	3 2	3 2 2	3 2 2	3	3 2	3 2 1

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'
```

act3scene10	1 1 1 2 1 1 {."0 1 act3scene10
Thou shalt remain here, whether thou wilt or no. I am a spirit of no common rate; The summer still doth tend upon my state; And I do love thee; therefore, go with me. I'll give thee fairies to attend on thee, And they shall fetch thee jewels from the deep.	T I T An I A
6 48	6 2

You can check the ranks of a verb `f` by `f b. 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'
```

$ b. 0	x	y	x$y	x$"1 y	x$"0 y	x$"0 0 y	x$"1 0 y
_ 1 _	2 3 3 5	5 6	5 6 5 0 0 6 5 6 0 0 0 0 0 0 0 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5	5 6 5 0 0 6 5 6 0 0 0 0 0 0 0 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5	5 5 0 0 0 5 5 5 0 0 6 6 6 0 0 6 6 6 6 6	5 5 0 0 0 5 5 5 0 0 6 6 6 0 0 6 6 6 6 6	5 5 5 0 0 5 5 5 0 0 0 0 0 0 0 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
3	2 2	2	2 3 5	2 3 5	2 2 5	2 2 5	2 3 5

All three ranks of a verb V may be specified, using $V"k\,k_1\,k_2$. 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'
```

c	,/c	,"0/c	,"1/c	,/"2 c	,"0 1/c	,"0 1/"2 c	,:/c	,:"1/c
abcd efgh ijkl mnop qrst uvwx	abcd efgh ijkl mnop qrst uvwx	am bn co dp eq fr gs ht iu jv kw lx	abcdmnop efghqrst ijkluvwx	abcdefghijkl mnopqrstuvwx	amnop bmnop cmnop dmnop eqrst fqrst gqrst hqrst iuvwx juvwx kuvwx luvwx	aeijkl bfijkl cgijkl dhijkl mquvwx nruvwx osuvwx ptuvwx	abcd efgh ijkl mnop qrst uvwx	abcd mnop efgh qrst ijkl uvwx
2 3 4	6 4	3 4 2	3 8	2 12	3 4 5	2 4 6	2 3 4	3 2 4

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'
```

ProbTab	+/ b. 0	+/ProbTab	+/"1 ProbTab	+/"2 ProbTab
0.181 0.111 0.078 0.054 0.054 0.072 0.024 0.024 0.032 0.041 0.111 0.218	_ _ _	0.205 0.135 0.11 0.095 0.165 0.29	0.37 0.18 0.08 0.37	0.235 0.165 0.15 0.065 0.135 0.25
2 2 3	3	2 3	2 2	2 3

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

```
mean=. +/ % #
A=. i. 2 3 4
xrs 'A'; 'mean b. 0'; 'mean A'; 'mean"1 A'; 'mean"2 A'
```

A	mean b. 0	mean A	mean"1 A	mean"2 A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	_ _ _	6 7 8 9 10 11 12 13 14 15 16 17	1.5 5.5 9.5 13.5 17.5 21.5	4 5 6 7 16 17 18 19
2 3 4	3	3 4	2 3	2 4

The arguments of a dyadic verb V must *agree*: if `x.` is a frame $\mathcal{F}_1$ of k_1-cells, and `y.` is a frame $\mathcal{F}_2$ of k_2-cells, where V has rank $(k_1\, k_2)$, then the shapes of $\mathcal{F}_1$ and $\mathcal{F}_2$ 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 `1`s.

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

```
   i234=. i. 2 3 4
   xrs 'i234'; '2 1*i234'; '(i. 2 3)*i234'; '(i. 4)*"1 i234'; '(i. 2 4)*"1 i234'
+-----------+-----------+---------------+------------+----------------+
|i234       |2 1*i234   |(i. 2 3)*i234  |(i. 4)*"1 i234|(i. 2 4)*"1 i234|
+-----------+-----------+---------------+------------+----------------+
| 0  1  2  3| 0  2  4  6|  0   0   0   0|0  1  4  9  | 0   1   4   9  |
| 4  5  6  7| 8 10 12 14|  4   5   6   7|0  5 12 21  | 0   5  12  21  |
| 8  9 10 11|16 18 20 22| 16  18  20  22|0  9 20 33  | 0   9  20  33  |
|           |           |               |            |                |
|12 13 14 15|12 13 14 15| 36  39  42  45|0 13 28 45  |48  65  84 105  |
|16 17 18 19|16 17 18 19| 64  68  72  76|0 17 36 57  |64  85 108 133  |
|20 21 22 23|20 21 22 23|100 105 110 115|0 21 44 69  |80 105 132 161  |
+-----------+-----------+---------------+------------+----------------+
|2 3 4      |2 3 4      |2 3 4          |2 3 4       |2 3 4           |
+-----------+-----------+---------------+------------+----------------+
   (i. 4) * i234      NB. 4 is not a prefix of 2 3 4
|length error
|   (i.4)    *i234
   (i. 2) *"1 i234    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:

```
   A=. i. 2 1 3 4
   XYZ=. 'xyz'"_
   xrs 'A';'XYZ A';'7"_ A';'7"0 A';'7"1 A';'7"2 A';'7 8"1 A';'7"_1 A';'''X''"0 A';'a:"1 A'
+-----------+-----+-----+-------+-----+-----+-------+------+-------+------+
|A          |XYZ A|7"_ A|7"0 A  |7"1 A|7"2 A|7 8"1 A|7"_1 A|'X'"0 A|a:"1 A|
+-----------+-----+-----+-------+-----+-----+-------+------+-------+------+
| 0  1  2  3|xyz  |7    |7 7 7 7|7 7 7|7    |7 8    |7 7   |XXXX   |++++  |
| 4  5  6  7|     |     |7 7 7 7|     |7    |7 8    |      |XXXX   |||||  |
| 8  9 10 11|     |     |7 7 7 7|7 7 7|     |7 8    |      |XXXX   |++++  |
|           |     |     |       |     |     |       |      |       |      |
|12 13 14 15|     |     |7 7 7 7|     |     |7 8    |      |XXXX   |++++  |
|16 17 18 19|     |     |7 7 7 7|     |     |7 8    |      |XXXX   |||||  |
|20 21 22 23|     |     |7 7 7 7|     |     |7 8    |      |XXXX   |++++  |
+-----------+-----+-----+-------+-----+-----+-------+------+-------+------+
|2 1 3 4    |3    |     |2 1 3 4|2 1 3|2 1  |2 1 3 2|2     |2 1 3 4|2 1 3 |
+-----------+-----+-----+-------+-----+-----+-------+------+-------+------+
```

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 (2×2) matrix A is given by `(%/ . *)`A

```
A=. 1+i.2 2 [ B=. ?. 3 3$20
det=. -/ . * [. pm=. +/ . * [. tr=. [/ . + [. or=. %/ . *
xrs 'A';'det A';'pm A';'tr A';'or A';'B';'det B';'pm B';'tr B';'<./ . + B'
```

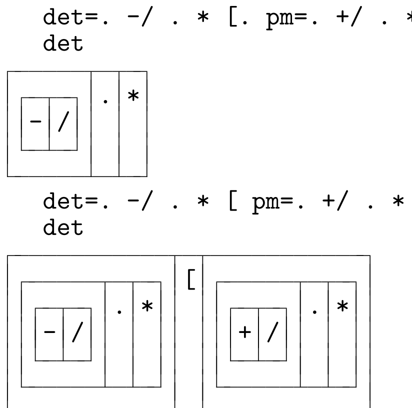
A	det A	pm A	tr A	or A	B	det B	pm B	tr B	<./ .+B
1 2 3 4	_2	10	5	0.6666667	2 15 9 10 4 0 13 13 18	_1854	4482	24	15
2 2					3 3				

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:

```
det=. -/ . * [. pm=. +/ . *
det
```



```
det=. -/ . * [ pm=. +/ . *
det
```

Similarly `].` (*dex*) is a conjunction that yields its right argument.

3. The expression `<./ .+B`, or equivalently `(<. / . +)B`, picks one atom from each row and each column of `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 `B`, are given by the columns of `,./ . ,. B`. Therefore `<./ +/ ,./ . ,. B` is `15` in the above example, and `+/ */ ,./ . ,. B` is `4482`, the permanent of `B`.

2.3.5.2 Matrix Multiplication (+/ . *)

Matrix multiplication in J is performed by dyadic `+/ . *`, an example of inner product (page 42).

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

```
A=. i. 3 4
mp=. +/ . *      NB. matrix product
xrs 'A';'A mp |:A';'(|:A) mp A';'x=.i.3';'y=.i.4';'x mp A';'A mp y';'y mp y'
```

<table>
<tr><td>A</td><td>A mp |:A</td><td>(|:A) mp A</td><td>x=.i.3</td><td>y=.i.4</td><td>x mp A</td><td>A mp y</td><td>y mp y</td></tr>
<tr><td>0 1 2 3
4 5 6 7
8 9 10 11</td><td>14 38 62
38 126 214
62 214 366</td><td> 80 92 104 116
 92 107 122 137
104 122 140 158
116 137 158 179</td><td>0 1 2</td><td>0 1 2 3</td><td>20 23 26 29</td><td>14 38 62</td><td>14</td></tr>
<tr><td>3 4</td><td>3 3</td><td>4 4</td><td>3</td><td>4</td><td>4</td><td>3</td><td></td></tr>
</table>

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

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}^n = \begin{pmatrix} F_{2n-1} & F_{2n} \\ F_{2n} & F_{2n+1} \end{pmatrix}, \tag{2.3}$$

where F_k is the kth Fibonacci number, can be demonstrated in J as follows:

```
A=. x: 1 2 2$1 1 1 2
f=. [: (+/ . *)/ #&A
xrs 'A'; 'f 2'; 'f 3'; 'f 4'; 'f 10'; 'f 50'
```

A	f 2	f 3	f 4	f 10	f 50
1 1 1 2	2 3 3 5	5 8 8 13	13 21 21 34	4181 6765 6765 10946	218922995834555169026 354224848179261915075 354224848179261915075 573147844013817084101
1 2 2	2 2	2 2	2 2	2 2	2 2

Note however that there are far more efficient ways to calculate A^n for large n, see Section 2.4.3.

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' ; '(mp %.) B'
```

B	%. B	(mp %.) B
1 9 5 6 2 0 8 8 12 4 6 10 0 0 6 8	_0.0334728 0.06485356 0.07531381 _0.1338912 0.1129707 _0.03138075 _0.0041841 _0.04811715 0.0334728 0.4351464 _0.07531381 _0.3661088 _0.0251046 _0.3263598 0.05648536 0.3995816	1 0 0 0 _2.77556e_17 1 0 4.44089e_16 5.55112e_17 0 1 8.88178e_16 _2.77556e_17 0 0 1
4 4	4 4	4 4

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 arithemtic operations, `%.` can be carried out to arbitrary precision:

```
   xrs 'B' ; '%. x: B' ; '(mp %.) x: B' ; '%. %. x: B'
```

<table>
<tr><td>B</td><td>%. x: B</td><td>(mp %.) x: B</td><td>%. %. x: B</td></tr>
<tr><td> 1 9 5 6
 2 0 8 8
12 4 6 10
 0 0 6 8</td><td>_8r239   31r478   18r239   _32r239
27r239 _15r478   _1r239   _23r478
 8r239 104r239 _18r239 _175r478
_6r239 _78r239   27r478   191r478</td><td>1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1</td><td> 1 9 5 6
 2 0 8 8
12 4 6 10
 0 0 6 8</td></tr>
<tr><td>4 4</td><td>4 4</td><td>4 4</td><td>4 4</td></tr>
</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 $r \geq c$), 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 $\widehat{y}_i = \widehat{\beta}_0 + \widehat{\beta}_1 x_i$ to the data

i	1	2	3	4	5
x_i	1	3	4	5	7
y_i	8	8	4	4	1

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

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

In J, given `y` and `X` as above, the fitted parameter vector $\widehat{\boldsymbol{\beta}} = (\widehat{\beta}_0, \widehat{\beta}_1)$ 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
   X=. 1,.x
   xrs 'y'; 'X'; 'b=.y%.X'; 'Xt=.|:X'; '(%. Xt mp X) mp Xt mp y'; 'X mp b'
```

<table>
<tr><td>y</td><td>X</td><td>b=.y%.X</td><td>Xt=.|:X</td><td>(%. Xt mp X) mp Xt mp y</td><td>X mp b</td></tr>
<tr><td>8 8 4 4 1</td><td>1 1
1 3
1 4
1 5
1 7</td><td>10 _1.25</td><td>1 1 1 1 1
1 3 4 5 7</td><td>10 _1.25</td><td>8.75 6.25 5 3.75 1.25</td></tr>
<tr><td>5</td><td>5 2</td><td>2</td><td>2 5</td><td>2</td><td>5</td></tr>
</table>

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.

```
xrs 'A=. 1,.i.4';'X=.%.A';'A mp X mp A';'X mp A mp X';'A mp X';'X mp A'
```

A=. 1,.i.4	X=.%.A	A mp X mp A	X mp A mp X	A mp X	X mp A
1 0 1 1 1 2 1 3	0.7 0.4 0.1 _0.2 _0.3 _0.1 0.1 0.3	1 0 1 1 1 2 1 3	0.7 0.4 0.1 _0.2 _0.3 _0.1 0.1 0.3	0.7 0.4 0.1 _0.2 0.4 0.3 0.2 0.1 0.1 0.2 0.3 0.4 _0.2 0.1 0.4 0.7	1 0 0 1
4 2	2 4	4 2	2 4	4 4	2 2

3. If `y.` is a nonzero vector, then `%.y.` inverts it with respect to the unit circle; see page 35.

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 `g` dyadically between corresponding elements, and then `f` monadically to the resulting vector.

In practice, the verb `f` in the inner product `f.g` is usually derived by insertion, for example
`x. (+/ . =) y.` counts the matches between each row of `x.` & column of `y.`,
`x. (*./ . =) y.` returns `1` for each row of `x.` and column of `y.` that are identical, and
`x. (+./ . ~:) 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 mory simply produced by other J verbs such as `-:` (*match*) or `e.` (*member*):

```
x=. 5 3$'catbatmanpigemu' [ y=. 3 3$'antmanbee'
xrs 'x';'y';'x +/ . = |:y';'x *./ . =|: y';'x +./ . ~: |:y';'x -:"1/ y';'y e. x'
```

<table>
<tr><th>x</th><th>y</th><th>x +/ . = |:y</th><th>x *./ . =|: y</th><th>x +./ . ~: |:y</th><th>x -:"1/ y</th><th>y e. x</th></tr>
<tr><td>cat
bat
man
pig
emu</td><td>ant
man
bee</td><td>1 1 0
1 1 1
0 3 0
0 0 0
0 0 0</td><td>0 0 0
0 0 0
0 1 0
0 0 0
0 0 0</td><td>1 1 1
1 1 1
1 0 1
1 1 1
1 1 1</td><td>0 0 0
0 0 0
0 1 0
0 0 0
0 0 0</td><td>0 1 0</td></tr>
<tr><td>5 3</td><td>3 3</td><td>5 3</td><td>5 3</td><td>5 3</td><td>5 3</td><td>3</td></tr>
</table>

Inner products can also be applied to vectors, as illustrated on page 40 for matrix product and in the following example:

```
   xrs '2 3 37 */ .^ 1 3 1';'factors=. __ q: 666 1998 999999';'(*/ .^)/"2 factors'
+------------------+-------------------------------+------------------+
|2 3 37 */ .^ 1 3 1|factors=. __ q: 666 1998 999999|(*/ .^)/"2 factors|
+------------------+-------------------------------+------------------+
|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                 |
+------------------+-------------------------------+------------------+
```

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):

```
   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 1)*"1 -/ ?. 2 7 4$8 15 10 40   NB. simulate marks
   xrs 'M';'max,:pass';'M *./ . >: pass';'100 * M <./ . % max';'M (i.>./) . % max'
+-----------+-------------+---------------+--------------------+-----------------+
|M          |max,:pass    |M *./ . >: pass|100 * M <./ . % max |M (i.>./) . % max|
+-----------+-------------+---------------+--------------------+-----------------+
|40 33 34 61|100 50 60 100|1 0 1 0 0 1 1  |40 48 52 25 40 64 47|1 2 3 2 1 2 2    |
|65 24 46 52| 40 25 30  35|               |                    |                 |
|65 26 44 74|             |               |                    |                 |
|25 25 36 50|             |               |                    |                 |
|60 31 24 56|             |               |                    |                 |
|70 32 50 74|             |               |                    |                 |
|70 31 50 47|             |               |                    |                 |
+-----------+-------------+---------------+--------------------+-----------------+
|7 4        |2 4          |7              |7                   |7                |
+-----------+-------------+---------------+--------------------+-----------------+
```

The final example of inner product illustrates the more general case `x. f . g 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) +/ 0 0 0 0 0
   y=. #: 1 2 3 4,: 7 8 17 31
   xrs 'x' ; 'y' ; '#. b. 0' ; 'x < . #. y' ; 'x +/ . #. y' ; '2 3#."0 1 y'
+---------+---------+-------+------------------+----------------+------------+
|x        |y        |#. b. 0|x < . #. y        |x +/ . #. y     |2 3#."0 1 y |
+---------+---------+-------+------------------+----------------+------------+
|2 2 2 2 2|0 0 0 0 1|1 1 1  |+----------------+|14  29   85  125| 1  2  3   4|
|3 3 3 3 3|0 0 0 1 0|       || 1  2  3   4    ||                |13 27 82 121|
|         |0 0 0 1 1|       ||13 27 82 121    ||32 129  631  797|            |
|         |0 0 1 0 0|       |+----------------+|                |            |
|4 4 4 4 4|         |       || 1   4   5  16  ||58 349 2409 2837|            |
|5 5 5 5 5|0 0 1 1 1|       ||31 125 626 781  ||                |            |
|         |0 1 0 0 0|       |+----------------+|                |            |
|         |1 0 0 0 1|       || 1   6    7   36||                |            |
|6 6 6 6 6|1 1 1 1 1|       ||57 343 2402 2801||                |            |
|7 7 7 7 7|         |       |+----------------+|                |            |
+---------+---------+-------+------------------+----------------+------------+
|3 1 2 5  |2 4 5    |3      |3 1               |3 1 4           |2 4         |
+---------+---------+-------+------------------+----------------+------------+
```

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 `}.$y.`) in a frame of shape `}:$x.`.

2.3.5.5 Miscellaneous Matrix Functions

One simple way to produce the `n`×`n` *identity matrix* is `(=@i.)n`, using the verb *self-classify*. Many other highly-structured matrices, such as triangular matrices, Hilbert matrices and Toeplitz matrices[8], are easily produced via the *table* adverb (page 50):

```
ident=. = @ i.
lowertri=. >:/~ @ i.
hilbert=. % @ >: @ (+/~) @ i.
toeplitz=. ((<: - i. -/ i.)@>.@-:@#) { ]
xrs 'ident 5'; 'lowertri 5'; 'hilbert 5x'; 'toeplitz 31 41 59 26 53 58 97 93 23'
```

ident 5	lowertri 5	hilbert 5x	toeplitz 31 41 59 26 53 58 97 93 23
1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	1 0 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1 0 1 1 1 1 1	1 1r2 1r3 1r4 1r5 1r2 1r3 1r4 1r5 1r6 1r3 1r4 1r5 1r6 1r7 1r4 1r5 1r6 1r7 1r8 1r5 1r6 1r7 1r8 1r9	53 58 97 93 23 26 53 58 97 93 59 26 53 58 97 41 59 26 53 58 31 41 59 26 53
5 5	5 5	5 5	5 5

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 `&` (*bond*) glues the noun to the verb as one of its arguments, giving a new (monadic) verb:

```
   log10=. 10&^.
   cube=. ^&3
   cuberoot=. 3&%:
   twopower=. 2&^
   xrs 'log10 2 3 4 100' ; 'cube 2 3 5' ; 'cuberoot 10 125' ; 'twopower i. 6'
```

log10 2 3 4 100	cube 2 3 5	cuberoot 10 125	twopower i. 6
0.30103 0.4771213 0.60206 2	8 27 125	2.15443 5	1 2 4 8 16 32
4	3	2	6

Between two monadic verbs, `&` denotes *composition*: `f&g y.` means `f(g(y.))`, i.e. ‘apply `g` to `y.`, then `f` to the result’. In the dyadic case, `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:

```
   isanag=: -: & (/:~)              NB. (dyadic) is x. an anagram of y.?
   'integral calculus' isanag 'calculating rules'
1
   'eleven + two' isanag 'twelve + one'
1
   'punishment' isanag 'nine thumps'
0
   noblanks=: -.&' '                NB. (monadic) remove blanks from y.
   isanag1=: isanag & noblanks
   'punishment' isanag1 'nine thumps'
1
   'schoolmaster' isanag1 'the classroom'
1
   'the aristocracy' isanag1 'a rich tory caste'
1
   'Anagrams' isanag1 'Ars magna'
1
   'Horatio Nelson' isanag1 'Honor est a Nilo'
1
   'incomprehensible' isanag1 'problem in Chinese'
0
   a. i. 'aAZ'
97 65 90
   isupper=: >:&65 *. <:&90
   lfu=: + (32&* & isupper)         NB. lower-case index from upper-case index
   lowercase=: lfu &. (a.&i.)       NB. convert upper-case to lower
   isanag2=: isanag1 & lowercase
   'incomprehensible' isanag2 'problem in Chinese'
1
   'Ivanhoe by Sir Walter Scott' isanag2 'A novel by a Scottish writer'
1
   'Nessiteras Rhombopteryx' isanag2 'Monster hoax by Sir Peter S.'
0
   islower=: >:&97 *. <:&122
   lowercase=: ((islower # ]) & lfu) &. (a.&i.)     NB. redefined
   'Nessiteras Rhombopteryx' isanag2 'Monster hoax by Sir Peter S.'
1
   'King''s Lead Hat' isanag2 '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'
1
```

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 (`isanag`) fails if the number of spaces in the left and right arguments don't agree. The second attempt (`isanag1`) corrects this, but takes no account of upper- and lower-case letters. The third attempt (`isanag2`) 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"_ * isupper)` could be used instead of `(32&* & isupper)`.
4. 'Ars magna' means 'great art', 'Nessiteras 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 $-\frac{2}{5}$.

Examples of dyadic `&` and `@` follow:

```
   a=. 1 5
   b=. i. 2 2
   xrs '2 -&% 5';'2 -@% 5';'a';'b';'a ,@$ b';'a ,&$ b';'a $@, b';'a $&, b';'a$b';'a,b'
```

2 -&% 5	2 -@% 5	a	b	a ,@$ b	a ,&$ b	a $@, b	a $&, b	a$b	a,b
0.3	_0.4	1 5	0 1 2 3	0 1 2 3 0 1 2 3 0 1	2 2 2	3 2	0 1 2 3 0	0 1 2 3 0 1 2 3 0 1	1 5 0 1 2 3
		2	2 2	10	3	2	1 5	1 5 2	3 2

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
```

2.3.6.2 Power (`^:`)

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

```
   (*: ^: 2) 0 1 2 3    NB. ((0^2)^2) , ((1^2)^2) , ((2^2)^2), ((3^2)^2)
0 1 16 81
   (*: ^: 0 1 2 3) 2    NB. 2 , (2^2) , ((2^2)^2) , (((2^2)^2)^2)
2 4 16 256
   (^~ ^: 2) 2x 3x      NB. ((2^2)^(2^2)) , ((3^3)^(3^3))
256 443426488243037769948249630619149892803
```

J recognises when iterated applications of a verb converge: the expression `f^:_ 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^:_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 $\sqrt{3}$:

```
   nxt=. -: @ (+ 3&%)     NB. next approximation to square root of 3
   xrs 'nxt 1';'nxt 1.5';'nxt^:2(1)';'nxt^:3(1)';'x=.nxt^:_(1)';'x^2';'*:^:_1(3)'
```

nxt 1	nxt 1.5	nxt^:2(1)	nxt^:3(1)	x=.nxt^:_(1)	x^2	*:^:_1(3)
2	1.75	1.75	1.73214	1.73205	3	1.73205

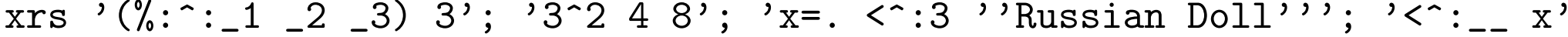
This is easily, if unnecessarily, generalised to a function to find the square root of any number. The following example finds $\sqrt{2}$ and $\sqrt{3}$ to 14 significant figures, and shows that $9^2 + 19^2/22$ is approximately π^4:

```
   pp=. (9!:10) 0         NB. save current print precision
   (9!:11) 14             NB. increase print precision

   nxt=. -: @ (% + ])     NB. next approximation from y. to sqrt(x.)
   sqrt=. ] nxt^:_ 1:     NB. create verb 'sqrt' equivalent to %:
   xrs '3 nxt 1';'3(nxt^:_)1';'sqrt 2';'sqrt^:2 (*:9)+(*:19)%22';'1p1'
```

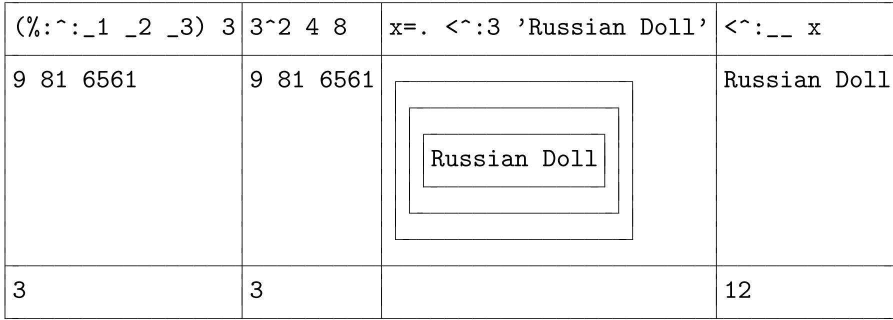
3 nxt 1	3(nxt^:_)1	sqrt 2	sqrt^:2 (*:9)+(*:19)%22	1p1
2	1.7320508075689	1.4142135623731	3.1415926525826	3.1415926535898

```
   9!:11 pp               NB. reset print precision
```

Finally, the expression `f^:n` where `n` is a *negative* integer means 'apply the inverse of `f`, `n` times':



```
   xrs '(%:^:_1 _2 _3) 3'; '3^2 4 8'; 'x=. <^:3 ''Russian Doll'''; '<^:__ x'
```

(%:^:_1 _2 _3) 3	3^2 4 8	x=. <^:3 'Russian Doll'	<^:__ x
9 81 6561	9 81 6561	 Russian Doll	Russian Doll
3	3		12

2.3.6.3 Under (`&.`)

The expression `f&.g` ('`f` under `g`') is like `f&g`, except that `g^:_1` is applied to each cell of the result. Note that transformations of the form $g^{-1}fg$ 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.`':

```
   I=. ^: _1
   xrs '*:13'; '*:I 169'; '+:&.^. 13'; 'p:i.10'; 'p:I 1000000'; '<:&.(p:I) 1000000'
```

`*: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
			10		

```
   p: 78497 78498
999983 1000003
```

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 `stdlib` 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'
   y
```

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

```
   |.each y    NB. i.e. |.&.> y
```

4 3 2 1 0	5 6 7 8 9 0 1 2 3 4	second first	animal	ablE was I ere I saw elbA

```
   |. y        NB. spot the difference
```

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

J can deduce the inverses of many other bijective operations, implicitly using rules like $(fg)^{-1} = g^{-1}f^{-1}$. 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.
   iflmf=: _1: + +:@isheadmax\.          NB. multiplying factor for each char
   ifl=: +/ @ (* iflmf) @ iflchar        NB. Integer from Latin numerals
   ifl 'MCMLIX'
1959
   lfi=: _ 2 5 2 5 2 5&#: # 'MDCLXVI'"_   NB. partial inverse (IIII not IV etc.)
   lfi 1959
MDCCCCLVIIII
```

```
   lfi # 'Iosephus Agrestis Piscesque'
XXVII
   IFL=: ifl :. lfi                       NB. OBVERSE: use lfi as inverse to ifl
   inLatin=: &. IFL
   'III' *inLatin 'DCLIII'
MDCCCCLVIIII
   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 necessary. The customised verb `f!.a` applies `f`, padding out with the atom `a` as needed:

```
   i23=. i. 2 3 [ i5=. i. 5 [ a23=. 2 3$'abcdef' [ a5=. 5$'pqrst'
   xrs 'i23,i5';'i23 ,!._1 i5';'a23,a5';'a23 ,!.''-'' a5';'3 6$i5';'3 6$!.99 i5'
```

i23,i5	i23 ,!._1 i5	a23,a5	a23 ,!.'-' a5	3 6$i5	3 6$!.99 i5
0 1 2 0 0 3 4 5 0 0 0 1 2 3 4	0 1 2 _1 _1 3 4 5 _1 _1 0 1 2 3 4	abc def pqrst	abc-- def-- pqrst	0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2	0 1 2 3 4 99 99 99 99 99 99 99 99 99 99 99 99 99
3 5	3 5	3 5	3 5	3 6	3 6

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           NB. 'tolerably equal', by default to 1 in 2^_44 (roughly 5e_14)
0 0 0 0 1 1 1 1 1 1
   y (=!.0) 1      NB. equal up to computer precision (roughly 1 in 1e_15)
0 0 0 0 0 0 1 1 1 1
   exactly=. !.0   NB. possibly useful adverb
   y =exactly 1
0 0 0 0 0 0 1 1 1 1
   1 | y           NB. result is rounded to 0 if tolerably close
1e_10 1e_11 1.00009e_12 9.99201e_14 0 0 0 0 0 0
   1 |exactly y
1e_10 1e_11 1.00009e_12 9.99201e_14 9.99201e_15 1.11022e_15 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`.

2.3.7 Miscellaneous Constructions

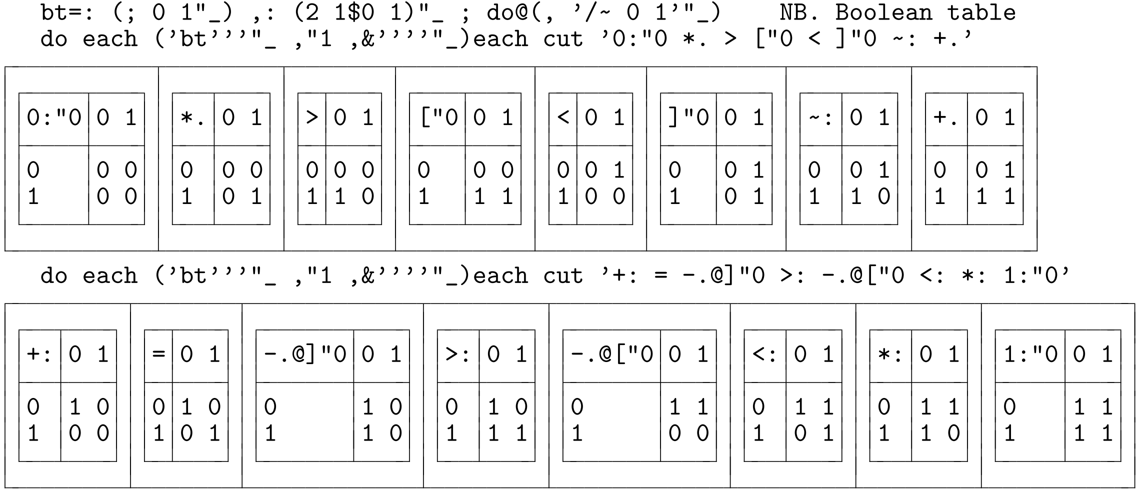
2.3.7.1 Table (Dyadic `u. /`) and Outer Products

If the verb `f` has dyadic rank $k_1\ k_2$, then `x. f/ y.` creates a 'function table' showing the results of applying `f` between each k_1-cell of `x.` and each k_2-cell of `y.`. For example, `x. +/ y.` and `x. */ y.` produce an addition table and a multiplication table respectively. Such function tables are often useful when learning or checking the action of verbs:

```
xrs '+/~ i.6'; '*/~ i.6'; '-/~ i.6'; '(i.7)^/i.5'; '''N S'' ,&.>"0/ ''W E'''
```

```
+---------------+----------------+-------------------+-----------------+----------------------+
|+/~ i.6        |*/~ i.6         |-/~ i.6            |(i.7)^/i.5       |'N S' ,&.>"0/ 'W E'   |
+---------------+----------------+-------------------+-----------------+----------------------+
|0 1 2 3 4  5   |0 0  0  0  0  0 |0 _1 _2 _3 _4 _5   |1 0  0   0    0  |+--+-+--+             |
|1 2 3 4 5  6   |0 1  2  3  4  5 |1  0 _1 _2 _3 _4   |1 1  1   1    1  ||NW|N|NE|             |
|2 3 4 5 6  7   |0 2  4  6  8 10 |2  1  0 _1 _2 _3   |1 2  4   8   16  |+--+-+--+             |
|3 4 5 6 7  8   |0 3  6  9 12 15 |3  2  1  0 _1 _2   |1 3  9  27   81  || W| | E|             |
|4 5 6 7 8  9   |0 4  8 12 16 20 |4  3  2  1  0 _1   |1 4 16  64  256  |+--+-+--+             |
|5 6 7 8 9 10   |0 5 10 15 20 25 |5  4  3  2  1  0   |1 5 25 125  625  ||SW|S|SE|             |
|               |                |                   |1 6 36 216 1296  |+--+-+--+             |
+---------------+----------------+-------------------+-----------------+----------------------+
|6 6            |6 6             |6 6                |7 5              |3 3                   |
+---------------+----------------+-------------------+-----------------+----------------------+
```

Similarly, there are 16 (i.e. 2^{2*2}) possible dyadic Boolean functions; J expressions for them and tables verifying their effects are conveniently displayed using the table adverb:



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

```
+-----------+---------+-------+----------+-------+----------+---------+---------+
|+---+---+  |+--+---+ |+-+---+|+---+---+ |+-+---+|+---+---+ |+--+---+ |+--+---+ |
||0:"0|0 1| ||*.|0 1| ||>|0 1|||["0|0 1| ||<|0 1|||]"0|0 1| ||~:|0 1| ||+.|0 1| |
|+---+---+  |+--+---+ |+-+---+|+---+---+ |+-+---+|+---+---+ |+--+---+ |+--+---+ |
||0  |0 0|  ||0 |0 0| ||0|0 0|||0  |0 0| ||0|0 1|||0  |0 1| ||0 |0 1| ||0 |0 1| |
||1  |0 0|  ||1 |0 1| ||1|1 0|||1  |1 1| ||1|0 0|||1  |0 1| ||1 |1 0| ||1 |1 1| |
|+---+---+  |+--+---+ |+-+---+|+---+---+ |+-+---+|+---+---+ |+--+---+ |+--+---+ |
+-----------+---------+-------+----------+-------+----------+---------+---------+
```

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

```
+---------+-------+-----------+---------+-----------+---------+---------+-----------+
|+--+---+ |+-+---+|+------+---+|+--+---+ |+------+---+|+--+---+ |+--+---+ |+----+---+ |
||+:|0 1| ||=|0 1|||-.@]"0|0 1|||>:|0 1| ||-.@["0|0 1|||<:|0 1| ||*:|0 1| ||1:"0|0 1| |
|+--+---+ |+-+---+|+------+---+|+--+---+ |+------+---+|+--+---+ |+--+---+ |+----+---+ |
||0 |1 0| ||0|1 0|||0     |1 0|||0 |1 0| ||0     |1 1|||0 |1 1| ||0 |1 1| ||0   |1 1| |
||1 |0 0| ||1|0 1|||1     |1 0|||1 |1 1| ||1     |0 0|||1 |0 1| ||1 |1 0| ||1   |1 1| |
|+--+---+ |+-+---+|+------+---+|+--+---+ |+------+---+|+--+---+ |+--+---+ |+----+---+ |
+---------+-------+-----------+---------+-----------+---------+---------+-----------+
```

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

As a third example, the *circular functions* (sine, cosine, tangent, arctan etc.) are implemented in J by dyadic `o.`, whose right argument picks the circular function required (`1`=sine, `_1`=arcsine, etc.) The following table shows $\sin(\theta)$, $\cos(\theta)$ & $\tan(\theta)$ for $\theta = 0°, 30°, 45°, 60°, 90°, 135°, 180°, 270°, 360°$:

```
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
```

```
+---+-------------------------------------------------------------------------------+
|deg|0        30        45        60          90       135 180 270           360     |
|sin|0       0.5 0.7071068 0.8660254           1 0.7071068   0  _1 _2.44921e_16     |
|cos|1 0.8660254 0.7071068       0.5 6.12303e_17 _0.7071068 _1   0            1     |
|tan|0 0.5773503         1   1.73205  1.63318e16         _1   0  __ _2.44921e_16     |
+---+-------------------------------------------------------------------------------+
```

Many familiar mathematical objects, for example Pascal’s triangle, identity matrices & Hilbert matrices, can be simply expressed using the table adverb reflexively (see also page 44):

```
   xrs ’!/~ i.5’ ; ’=/~ i.6’ ; ’%@>:@(+/~) i.4’ ; ’<:/~ i.5’
```

!/~ i.5	=/~ i.6	%@>:@(+/~) i.4	<:/~ i.5
1 1 1 1 1 0 1 2 3 4 0 0 1 3 6 0 0 0 1 4 0 0 0 0 1	1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1	1 0.5 0.3333333 0.25 0.5 0.3333333 0.25 0.2 0.3333333 0.25 0.2 0.1666667 0.25 0.2 0.1666667 0.1428571	1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 0 1
5 5	6 6	4 4	5 5

Finite fields and related algebraic and combinatorial structures can similarly be represented and manipulated via the table adverb

```
   mt=. (| */~@}.@i.)@p:      NB. multiplication table for prime-order Finite Field
   xrs ’p: 2 3 4’ ; ’mt 2’ ; ’mt 3’ ; ’mt 4’
```

p: 2 3 4	mt 2	mt 3	mt 4
5 7 11	1 2 3 4 2 4 1 3 3 1 4 2 4 3 2 1	1 2 3 4 5 6 2 4 6 1 3 5 3 6 2 5 1 4 4 1 5 2 6 3 5 3 1 6 4 2 6 5 4 3 2 1	1 2 3 4 5 6 7 8 9 10 2 4 6 8 10 1 3 5 7 9 3 6 9 1 4 7 10 2 5 8 4 8 1 5 9 2 6 10 3 7 5 10 4 9 3 8 2 7 1 6 6 1 7 2 8 3 9 4 10 5 7 3 10 6 2 9 5 1 8 4 8 5 2 10 7 4 1 9 6 3 9 7 5 3 1 10 8 6 4 2 10 9 8 7 6 5 4 3 2 1
3	4 4	6 6	10 10

2.3.7.2 Catalogue (monadic `{`) and Cartesian Products

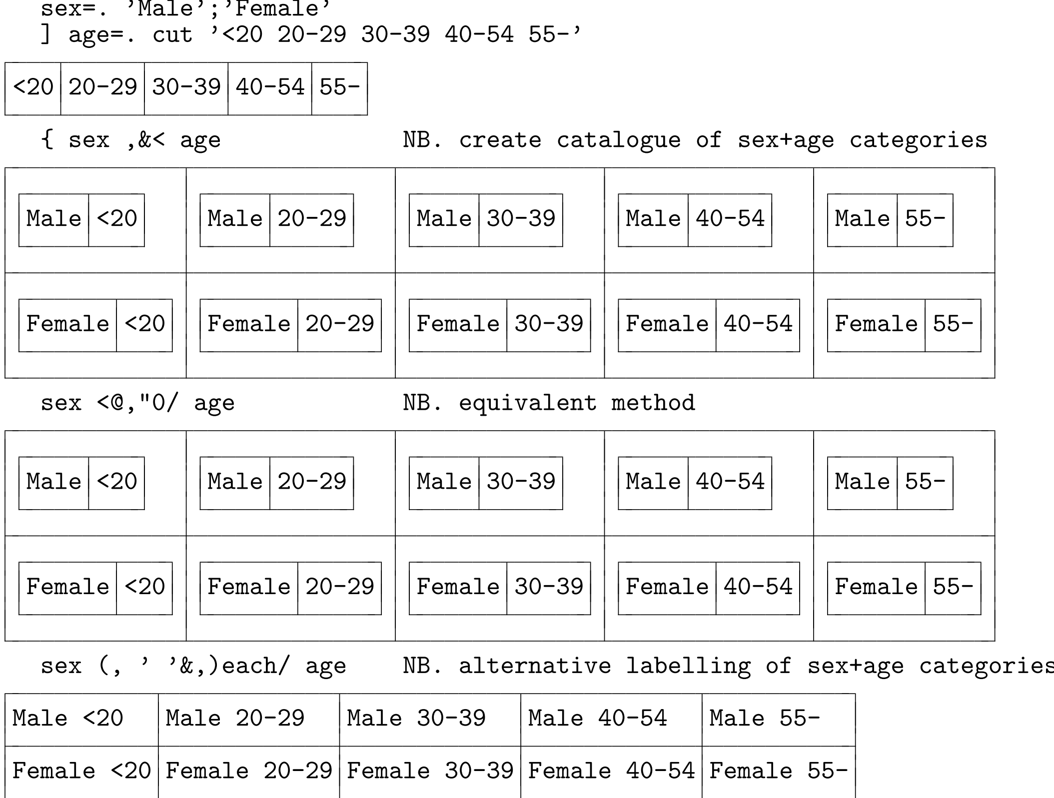
Monadic `{` creates a *catalogue* from the atoms in the opened items of its argument `y.`, as illustrated below:

```
   xrs ’{’’CDHS’’;’’A23456789TJQK’’’; ’{cut’’bl ei dgt’’’; ’{cut’’b aeiou dgt’’’
```

{’CDHS’;’A23456789TJQK’	{cut’bl ei dgt’	{cut’b aeiou dgt’
CA C2 C3 C4 C5 C6 C7 C8 C9 CT CJ CQ CK DA D2 D3 D4 D5 D6 D7 D8 D9 DT DJ DQ DK HA H2 H3 H4 H5 H6 H7 H8 H9 HT HJ HQ HK SA S2 S3 S4 S5 S6 S7 S8 S9 ST SJ SQ SK	bed beg bet bid big bit led leg let lid lig lit	bad bag bat bed beg bet bid big bit bod bog bot bud bug but
4 13	2 2 3	1 5 3

Thus if `y.` is a vector of boxes whose contents are the same type, then the result of `{y.` is a frame of shape `, $&> y.` of boxes, each box containing a vector of shape `$y.`.

Monadic `{` might be useful in analysing multi-way tables, for example to form labels for categories when a group of people is, so to speak, broken down by age and sex:



```
   sex=. 'Male';'Female'
   ] age=. cut '<20 20-29 30-39 40-54 55-'
┌───┬─────┬─────┬─────┬───┐
│<20│20-29│30-39│40-54│55-│
└───┴─────┴─────┴─────┴───┘
   { sex ,&< age            NB. create catalogue of sex+age categories
┌──────────┬────────────┬────────────┬────────────┬──────────┐
│┌────┬───┐│┌────┬─────┐│┌────┬─────┐│┌────┬─────┐│┌────┬───┐│
││Male│<20│││Male│20-29│││Male│30-39│││Male│40-54│││Male│55-││
│└────┴───┘│└────┴─────┘│└────┴─────┘│└────┴─────┘│└────┴───┘│
├──────────┼────────────┼────────────┼────────────┼──────────┤
│┌──────┬───┐│┌──────┬─────┐│┌──────┬─────┐│┌──────┬─────┐│┌──────┬───┐│
││Female│<20│││Female│20-29│││Female│30-39│││Female│40-54│││Female│55-││
│└──────┴───┘│└──────┴─────┘│└──────┴─────┘│└──────┴─────┘│└──────┴───┘│
└──────────┴────────────┴────────────┴────────────┴──────────┘
   sex <@,"0/ age           NB. equivalent method
┌──────────┬────────────┬────────────┬────────────┬──────────┐
│┌────┬───┐│┌────┬─────┐│┌────┬─────┐│┌────┬─────┐│┌────┬───┐│
││Male│<20│││Male│20-29│││Male│30-39│││Male│40-54│││Male│55-││
│└────┴───┘│└────┴─────┘│└────┴─────┘│└────┴─────┘│└────┴───┘│
├──────────┼────────────┼────────────┼────────────┼──────────┤
│┌──────┬───┐│┌──────┬─────┐│┌──────┬─────┐│┌──────┬─────┐│┌──────┬───┐│
││Female│<20│││Female│20-29│││Female│30-39│││Female│40-54│││Female│55-││
│└──────┴───┘│└──────┴─────┘│└──────┴─────┘│└──────┴─────┘│└──────┴───┘│
└──────────┴────────────┴────────────┴────────────┴──────────┘
   sex (, ' '&,)each/ age   NB. alternative labelling of sex+age categories
┌──────────┬────────────┬────────────┬────────────┬──────────┐
│Male <20  │Male 20-29  │Male 30-39  │Male 40-54  │Male 55-  │
├──────────┼────────────┼────────────┼────────────┼──────────┤
│Female <20│Female 20-29│Female 30-39│Female 40-54│Female 55-│
└──────────┴────────────┴────────────┴────────────┴──────────┘
```

2.3.7.3 Gerunds

A *gerund* is a verb acting as a noun; the idea is implemented in J by primitives such as `` ` `` (*tie*), `/` (*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:           NB. factorial 0
   fn=: * f@:<:      NB. factorial y.=n>0 defined recursively
   fa=: *            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 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=: &.>`

`inverse=: ^:_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

⇐

self-reference (`$:`)

`f^:g`, `f^:g^:_`,

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`:

```
   inv=: ^:_1              NB. (adverb): inverse of u.
   nplt=: p: inv           NB. number of primes less than y.
   sub10=: -&1 0           NB. subtract 1, 0 from y.
   pnp=: sub10 &. nplt     NB. previous, next prime (i.e. primes bracketing y.)
   from=: -~               NB. x. from y. (i.e. y. minus x.)
   lenpint=: from/ @ pnp   NB. length of interval between previous & next primes
   lenpint 1000
12
   pnp 1000
997 1009
```

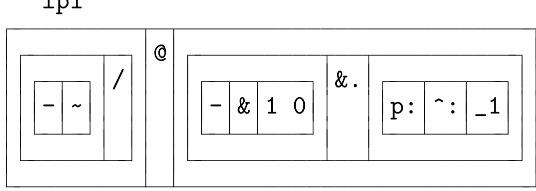
This shows that the length of the interval is 12, and that in fact the two primes bracketing 1000 are 997 and 1009.

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:

```
   lpi=:-~/@(-&1 0&.(p:^:_1))
   lpi 1000
12
```

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

```
   lpi
```

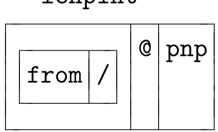


Note that by default J responds with the *boxed representation* of any expression that doesn't evaluate to a noun. This helps interpretation:

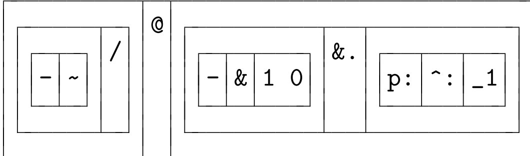
1. `lpi` is shown to be of the form V_1 `@` V_2.
2. Looking into the box representing V_2 shows that V_2 has the form 'V_{21} under the transformation V_{22}' (where V_{21} and V_{22} are further verbs).
3. V_{21} means 'subtract 1 0' and V_{22} means 'smallest prime $\geq$'. Therefore, with a composite argument, V_2 means 'previous, next prime' (you can confirm these with a little experimentation).
4. Finally, V_1 means 'insert `-~`'. In particular, if V_1s 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*):

```
   lenpint
```



```
   lenpint f.
```



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

```
   lpie=: 3 : 0
inp=: (p:^:_1) y. NB. index of next prime
pnp=: p: inp - 1 0
-~/ pnp
)
   lpie 1000
12
```

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

```
   lpi 1000 2000 3000
|length error: lpi
|      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 `y.`:

```
   (lpi"0) 1000 2000 3000
12 4 2
```

The zero-rank verbs `lenpint"0` and `lpie"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:

```
   sub10 nplt 1000 2000 3000
|length error: sub10
|       sub10 nplt 1000 2000 3000
   sub10"0 nplt 1000 2000 3000
167 168
302 303
429 430
```

Similarly, you need to change the rank of `-~/`. My preferred definition of `lpi`, which uses another possibly useful verb (`pnp`) and incorporates terse but helpful comments, would be:

```
   pnp=: -&1 0"0 &.(p:^:_1)  NB. previous, next prime for each atom of y.
   lpi=: -~/"1 @ pnp         NB. length of interval between previous & next primes
   lpi 1000 2000 3000
12 4 2
```

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:

```
   rfactab=: 1: ,. !@x:@i.         NB. 1st column 1, 2nd column e.p. factorials
   xrfr2=: (2&x:^:_1)"1            NB. e.p. rational from numerator, denom.
   erat=: +/ @ (xrfr2 @ rfactab)   NB. rational approx to e using y. terms
```

```
   e20=. erat 20
   (; x:^:_1) e20               NB. rational, real approx. to e
+-----------------------------+-------+
|82666416490601r30411275102208|2.71828|
+-----------------------------+-------+
   e50=. erat 50
   ft=: ": @ (<. @ (* 10x&^))      NB. format extended integer x. * 10^y.
   dp=: -@[ (}. ; {.) 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

⇐

`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 (`jl`) 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 Families of Foreign Conjunctions

Verbs defined by foreign conjunction can be divided into families according to the left argument of `!:` as outlined below.

3.2.1.1 Scripts (`0 !:`)

⇐

3.2.1.2 Files (`1 !:`)

⇐

3.2.1.3 Host Commands (`2 !:`)

These are provided primarily for Unix users; J for Windows provides much more extensive facilities using `11 !:` (see below).

3.2.1.4 Storage Types (`3 !:`)

⇐

3.2.1.5 Name Classes and Lists (`4 !:`)

⇐

```
   namelist noun
┌─────┬─┬────┬────┬─┬─┐
│dummy│g│misc│temp│v│x│
└─────┴─┴────┴────┴─┴─┘
   erase 'dummy temp x'
1 1 1
   namelist noun
┌─┬────┬─┐
│g│misc│v│
└─┴────┴─┘
```

3.2.1.6 Representations (`5 !:`)

⇐

3.2.1.7 Time (`6 !:`)

⇐

3.2.1.8 Space (`7 !:`)

⇐

```
   spacex=: 7!:2       NB. space used to execute expression
   spacex '1000#1'     NB. Boolean
2560
   spacex '1000#1x'    NB. integer
4672
   spacex '1000#1r1'   NB. rational
8864
   (spacex '1000#1.0') , spacex '1000#1.234'     NB. floating point
2560 8736
   (spacex '1000#1j0') , spacex '1000#1.234j1'   NB. complex
2560 17056
```

3.2.1.9 PC DOS Facilities (`8 !:`)

⇐

3.2.1.10 Global Parameters (`9 !:`)

⇐

Random link ⇐

```
   rlq=: 9!:0     NB. query random link
   rls=: 9!:1     NB. set random link
   rlq 0          NB. initial random seed
16807
   ? 5            NB. invoke J's PNG (pseudorandom number generator) once
0
   rlq 0          NB. updated value of random seed
282475249
   (7^5) ; (2^31) ; (2^31) | 7^10     NB. J's PNG is seed=: (2^31) | (7^5) * seed
+-----+----------+---------+
|16807|2147483648|282475249|
+-----+----------+---------+
   ? 20 # 100     NB. 20 pseudorandom 2-digit numbers produced by ? (roll)
75 45 53 21 4 67 67 93 38 51 83 3 5 52 67 0 38 6 41 68
   ? 20 # 100     NB. next set of 20 pseudorandom 2-digit numbers
58 93 84 52 9 65 41 70 91 76 26 4 73 32 63 75 99 36 24 98
   ?. 20 # 100    NB. 20 pseudorandom 2-digit numbers produced by ?. rather than ?
13 75 45 53 21 4 67 67 93 38 51 83 3 5 52 67 0 38 6 41
   ?. 20 # 100    NB. seed for ?. is reset each time to its initial value
13 75 45 53 21 4 67 67 93 38 51 83 3 5 52 67 0 38 6 41
   dummy=. rls 16807     NB. reset seed for ? to its initial value
   ? 20 # 100
13 75 45 53 21 4 67 67 93 38 51 83 3 5 52 67 0 38 6 41
```

Print precision ⇐

```
   ppq=: 9!:10    NB. query print precision
   pps=: 9!:11    NB. set print precision
   ppq 0
6
   dummy=. pps 14             NB. set printing precision to 14 s.f.
   (%: 1r3) ; -. %: 328r1835   NB. good approxns. to gamma = 0.577215664901532...
+----------------+----------------+
|0.577350269189626|0.577215664900591|
+----------------+----------------+
   dummy=. pps 6              NB. reset printing precision to previous value
```

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.

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**).

Full details are in the Release Notes [11].

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

⇐

[5]

- 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.jsoftware.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`

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

⇐

```
  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 3) ,.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=: , @ { @ (, & (<@:(<"1)))
NB.    (i.3 3) (>@(;&.>) @ cartbox) 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.    (i.20 3) ((>@(;&.>) @ cartbox) -: (,"1 cart)) i.20 3
NB. 1
```

```
NB.    20 timex '(i.20 3) (>@(;&.>) @ cartbox) i.20 3'
NB. 0.121
NB.    20 timex '(i.20 3) (,"1 cart) i.20 3'
NB. 0.0135
NB.    c1=. ,"1 cart f.
NB.    20 timex '(i.20 3) c1 i.20 3'
NB. 0.0085
```

4.1.2.2 copyshape

⇐

```
NB.
NB.    (i. 15) copyshape i. 3 4
NB. 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2
NB.    (i. 15) copyshape~ i. 3 4
NB. 0 1  2  3
NB. 4 5  6  7
NB. 8 9 10 11
```

4.1.2.3 expand

⇐

```
expand=: /:@\:@[ { #@[ {. ]

NB.
NB. (from Ken Iverson)
NB.
NB. ### Example ###
NB.    1 0 0 1 0 1 expand 2 3 4
NB. 2 0 0 3 0 4
```

4.1.2.4 expprod

⇐

```
expprod=:  (#~#) (($$(*+/\)@,)@[ { 0:,+/@,@[$]) ,@]

NB.
NB. ### Example + Comparison ###
NB.    expprod2=: shape2 @ (expand"1/)
NB.    x;y;(x expprod y);x expprod2 y
NB. ┌───────┬───┬───────┬───────┐
NB. │1 1 0 0│5 6│5 6 0 0│5 6 0 0│
NB. │1 0 1 0│7 8│7 8 0 0│7 8 0 0│
NB. │1 0 0 1│   │5 0 6 0│5 0 6 0│
NB. │0 1 1 0│   │7 0 8 0│7 0 8 0│
NB. │0 1 0 1│   │5 0 0 6│5 0 0 6│
NB. │0 0 1 1│   │7 0 0 8│7 0 0 8│
NB. │       │   │0 5 6 0│0 5 6 0│
NB. │       │   │0 7 8 0│0 7 8 0│
NB. │       │   │0 5 0 6│0 5 0 6│
NB. │       │   │0 7 0 8│0 7 0 8│
NB. │       │   │0 0 5 6│0 0 5 6│
NB. │       │   │0 0 7 8│0 0 7 8│
NB. └───────┴───┴───────┴───────┘
NB.    100 timex 'x expprod y'
NB. 0.0038
NB.    100 timex 'x expprod2 y'
NB. 0.0176
```

4.1.2.5 kronprod

⇐

```
kronprod=: *&$ $ ,@: (*"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
NB. ┌─────────┬───────────┐
NB. │0 1  2  3│10 11 12 13│
NB. │4 5  6  7│14 15 16 17│
NB. │8 9 10 11│           │
NB. └─────────┴───────────┘
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.    ($;]) shape2 4
NB. ┌───┬─┐
NB. │1 1│4│
NB. └───┴─┘
NB.    ($;]) shape2 i. 4
NB. ┌───┬───────┐
NB. │1 4│0 1 2 3│
NB. └───┴───────┘
NB.    ($;]) shape2 i. 3 4
NB. ┌───┬─────────┐
NB. │3 4│0 1  2  3│
NB. │   │4 5  6  7│
NB. │   │8 9 10 11│
NB. └───┴─────────┘
NB.    ($;]) shape2 i. 2 3 4
NB. ┌───┬───────────┐
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│
NB. └───┴───────────┘
```

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

```
   (shape2 -: ,.&.(|:"_1@|:)) i. 2 3 4 5 6 7
1
   100 timex 'shape2 i. 2 3 4 5 6 7'
0.0115
```

```
   100 timex ',.&.(|:"_1@|:) i. 2 3 4 5 6 7'
0.0434
```

4.1.2.8 diffs

⇐

```
diffs=: }. - }:

NB.
NB. ### Example ###
NB.    diffs 1 3 6 10
NB. 2 3 4
NB.    xy ; diffs xy
NB. ┌────┬────┐
NB. │ 0 0│1  2│
NB. │ 1 2│2  1│
NB. │ 3 3│3  0│
NB. │ 6 3│2 _1│
NB. │ 8 2│2 _1│
NB. │10 1│    │
NB. └────┴────┘
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
```

4.1.2.9 iotav

⇐

```
iotav=: [: ; i. each

NB.
NB. (from Roger Hui)
NB.
NB. ### Example ###
NB.   iotav 3 1 4 1 5
NB. 0 1 2 0 0 1 2 3 0 0 1 2 3 4
NB.
NB. ### JEHS unidiomatic alternative - was faster in J2! ###
NB.    timex '([: ; i.&.>) i.1000'     NB. J2 timing
NB. 3.46
NB.    timex '([: ; <@i."0) i.1000'    NB. J2 timing
NB. 3.08
NB.    10 timex '([: ; i.&.>) i.1000'  NB. J3
NB. 0.653
NB.    10 timex '([: ; <@i."0) i.1000' NB. J3
NB. 1.912
```

4.1.2.10 nubfreq

⇐

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

NB.
NB. ### Example ###
NB.    (/:~@~. ; nubfreq) 3 1 4 1 5 9 2 6 5 3 5
NB. ┌─────────────┬─────────────┐
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. ┌──────────────────────┬─────────
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=: <: @ (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=: <. @ (0.5&+)

NB.
NB. ### Example ###
NB.    round ((-: >: %: 5)^i.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 * -.@v2test) + v2true * v2test

NB.
NB. ### Example 1 ###
NB.    v2test =. >
NB.    v2true =. ]
NB.    v2false =. [
NB.    min =: v2 f.
NB.    3 1 4 1 5 9 min 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  =. >&9
NB.    v2true  =. -:
NB.    v2false =. +:
NB.    dorh1   =: v2 f.
NB. Quicker than the following
NB.    dorh =: +: ` -: @. (]>9:)
NB.    x =. ?5000$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*)

tabulation using `/.` (*key*)

monadic `?` (*roll*)

dyadic `?` (*deal*)

monadic `?.` (*roll (fixed seed)*)

dyadic `?.` (*deal (fixed seed)*)

J Phrases[2], chapter 10.

[20] [21] [22] [23] [24]

```
   gamma =: ! & <:
   beta =: *&gamma % gamma@+        NB. beta function B(x.,y.) from gamma function
   beta2=: % @ ((* % +) * ([ ! +))  NB. one way to get beta function from dyadic !
   0.7 beta 0.3
3.88322
   0.7 beta2 0.3
3.88322
   (! _0.3) * (! _0.7) % ! 0        NB. explicit calculation of B(0.7,0.3)
3.88322
   1p1 % 1 o. 0.3p1                 NB. B(x,1-x) = pi / sin(pi x) for 0<x<1
3.88322
```

4.3 Timetabling

This example illustrates

1. Creation and manipulation of a data-base in J,
2. Incorporating J output in a LaTeX document.

4.3.1 J Timetabling Script

The following J script (`timetab.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 LaTeX2e 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'
 'JEHS' ; < chop 'seminar ST104 ST104p ST217/b ST217s/b ST329 ST952'
 'JQS'  ; < chop 'ST114 ST301'
 'JW'   ; < chop 'ST318'
 'PJH'  ; < chop 'ST217/a ST217s/a ST304 ST329'
 'RJR'  ; < chop 'ST113 ST215 ST327'
 'SDJ'  ; < chop 'Stoch ST108 ST111/a ST111/b ST112/a ST112/b ST208 ST213'
 'WSK'  ; < chop 'ST202 ST333'
)
STAFF=: {."1 STAFFCOURSES
isstaff=: < e. STAFF&[

NB. ===================================================================
NB. Course code ; name; abbreviation ; size
NB.
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
 'ST108' ; 'Applications of Algebra and Analysis' ; 'AAA' ; 80
 'ST111' ; 'Probability (Part A)' ; 'Prob A' ; 350
 'ST111/a' ; 'Probability (Part A: Maths + M/S)' ; 'Prob A' ; 280
 'ST111/b' ; 'Probability (Part A: MORSE)' ; '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: MORSE)' ; 'Prob A' ; 70
 'ST113' ; 'Statistical Computing' ; 'Stat Comp' ; 75
```

< ... *Approximately 100 lines omitted* ... >

```
 'MA3F4' ; 'Linear Analysis' ; 'Lin An' ; 140
 'MA3G0' ; '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 @:> chop 0 : 0
 'CS117' ; 'minor'
 'CS123' ; 'tiny'
 'CS126' ; 'tiny'
 'CS128' ; 'minor'
 'EC106' ; 'core'
 'GE111' ; 'tiny'
 'IB104' ; 'core'
 'MA106' ; 'core'
 'MA112' ; 'tiny'
 'MA113' ; 'tiny'
 'MA125' ; 'minor'
 'MA128' ; 'tiny'
 'MA129' ; 'core'
 'MA130' ; 'minor'
 'MA131' ; 'core'
 'MA246' ; 'minor'
 'ST104' ; 'major'
 'ST105' ; 'minor'
 'ST108' ; 'core'
 'ST111/b' ; 'core'
 'ST112/b' ; 'core'
 'ST113' ; 'core'
 'ST114' ; 'major'
)

MORSE2=: do @:> chop 0 : 0
 'CS201' ; 'tiny'
```

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

```
MSC=: do @:> chop 0 : 0
 'Stoch' ; 'core'
 'STsem' ; '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 @:> 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 ; 'S021' ; 'w:1-10'
 'EC106'  ; 2 ; 'fri'  ; 16 ; 'S021' ; 'w:1-10'
```

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

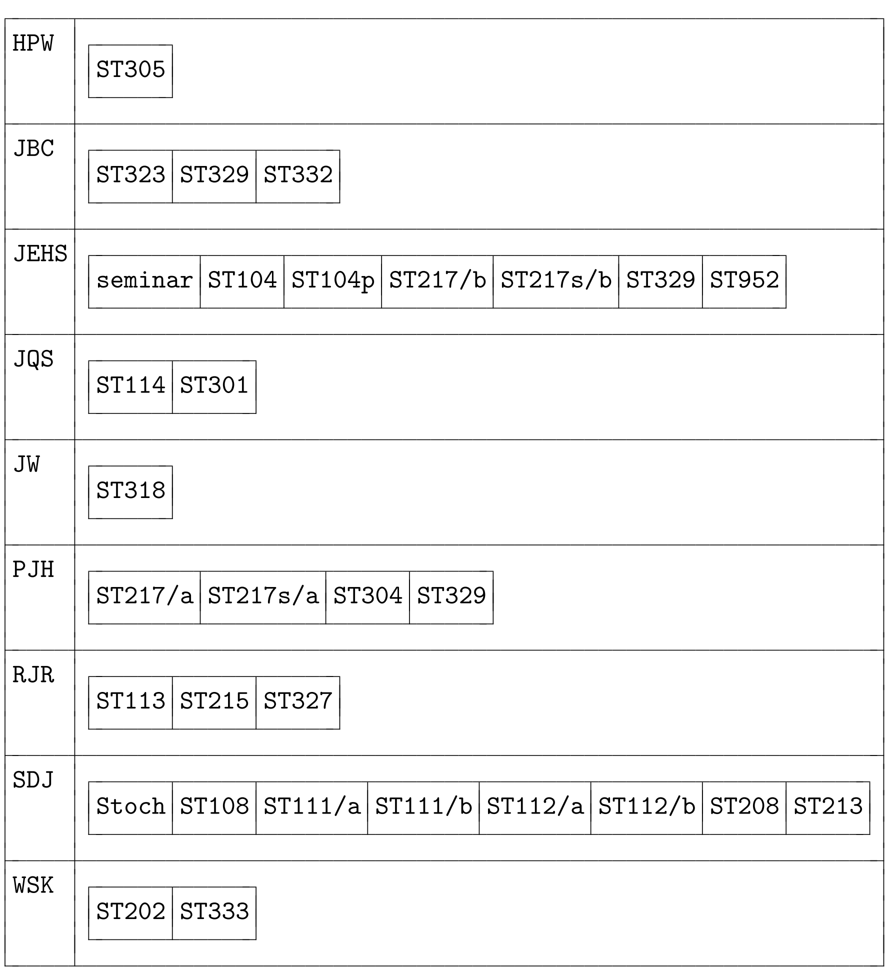
```
 'ST333'  ; 1 ; 'mon'  ; 12 ; 'rm68' ; 'w:2-10'
 'ST333'  ; 1 ; 'wed'  ; 11 ; 'rm68' ; 'w:1-10'
 'ST333'  ; 1 ; 'fri'  ; 11 ; 'B212' ; 'w:1-10'

 '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=. /:~ ~. (5&{.)each {."1 LECTURES
  list=. ((, 1{."1 COURSES) e."0 1 codes) # COURSES
  /:~ (> {."1 list) ,"1 ' : ' ,"1 (> 1{"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
  list=. i # LECTURES
  lt=. > 1 {"1 list
  (lt = term) # list
)

maketab1=: 3 : 0
  key=. TIMES i. > 3 {"1 y.
  }.each key </. y.
)

formatentry=: 3 : 0
  'c y d t r w' =. , y.
  ((i.&'/' {. ,&'/') c), ('  ',r) ,: 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=. > (maketab1)each table
)

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

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

```
TEXSEP=: LF, ('%',75#'='), LF, LF

setboldlist=: 3 : 0
  l=. (<'core') = {:"1 y.
  l # {."1 y.
)

tthead=: 3 : 0
  'start term' =. (}: ; {:) y.
  l1=. '\begin{center}', LF, ' {\Large\bf '
  l3=. ' Term ', term, '} \par \bigskip', LF
  if. (isstaff start) do.
    boldlist=: i. 0
    l2=. start
  else.
    do 'boldlist=: setboldlist ', start
    select. start
      case. 'MSC' do. l2=. 'M.Sc.'
      case. do.
        'course year' =. (}: ; {:) start
        select. course
          case. 'MS' do. l2=. 'Maths \& Stats'
          case. 'MORSE' do. l2=. 'MORSE'
          case. do. l2=. course
        end.
        l3=. ' Year ', year, l3
    end.
  end.
  l4=. '\begin{tabular}{|c|*{', (": #TIMES), '}{p{1.55cm}|}}', LF
  l5=. ' \tttimes', LF, '\\ \hline', LF
  l1,l2,l3,l4,l5
)

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

ttentry=: 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
  ' &', (0<#y.) # LF, (_7 }. vfm ttentry"1 y.), LF
)

ttrow=: 3 : 0
  t=. ((,&:>)/ ttslot each y.)
  t, ((LF ~: {:t) # LF), '\\ \hline', LF
)

ttbody=: 3 : 0
  ttdays=. 'Mon' ; 'Tues' ; 'Wed' ; 'Thurs' ; 'Fri'
  text=. ''
  for_d. i. 5 do.
    text=. text, ' \ttstrut ', (d pick ttdays), LF, ttrow d{y.
  end.
)

tt=:  3 : 0
  tth=. tthead y.
  ttb=. ttbody maketable lectlist y.
  tth,ttb,TTEND
)

cctab=: 3 : 0
  n=. >. -: # y.    NB. row after which to split codelist
  bc=. '{\small', LF, '\begin{center}', LF
  cc=. '{\Large\bf Course Codes} \par', LF, '\bigskip', LF
  ec=. '\end{center}', LF
  bt=. '\begin{tabular}{ll}', LF, ' '
  t1=. ((vfm n {. y.) rplc ' : ' ; ' & ') rplc LF ; ' \\',LF,' '
  t2=. ((vfm n }. y.) rplc ' : ' ; ' & ') rplc LF ; ' \\',LF,' '
  et=. '\end{tabular}', LF, '%', LF
  st=. '%', LF, '\hfill', LF, '%', LF
  ex=. '} % end{\small}', LF, '\newpage', '%', LF
```

```
  bc,cc,ec, bt,t1,et, st, bt,t2,et, ex
)

ALLTTH1=: 0 : 0
% Produced using \j304\jehs\misc\timetab.js
%
\documentclass[a4paper]{report}
\pagestyle{empty}
%
\setlength{\textheight}{25cm}
\setlength{\textwidth}{18cm}
\setlength{\topmargin}{-1cm}
\setlength{\oddsidemargin}{-1cm}
\setlength{\evensidemargin}{-1cm}
\setlength{\tabcolsep}{4pt}  % default 6pt
%
\newcommand{\ttentry}[3]
  {\rule{0pt}{13pt}\hfil {\large #1}\hfil\break\null\hfil
  {#2}\hfil\break\null\hfil \raisebox{3pt}{\scriptsize #3}\hfil}
\newcommand{\ttnov}[1]{\multicolumn{1}{c}{#1}}
\newcommand{\ttstrut}{\rule[-28pt]{0pt}{41pt}}
)

allttth2=: 3 : 0
  l1=. '\newcommand{\tttimes}{\ttnov{\rule[-7pt]{0pt}{1pt}} &', LF
  l2=. (": ,. TIMES) ,. '--' ,"1 (": ,. TIMES+1) ,"1 '} &', LF
  l1, (_2}. , ' \ttnov{' ,"1 l2), '}', LF
)

ALLTTHEAD=: ALLTTH1, (alltth2 0), TEXSEP, '\begin{document}', LF, '%', LF

alltt=: 3 : 0
  fname=. '\temp\alltt.tex'
  ('% All timetables ', (tstamp ''), LF) fwrites fname
  ALLTTHEAD fappends fname
  list=. chop 'MORSE1 MORSE2 MORSE3 MS1 MS2 MS3 MSC'
  for_name. list, STAFF do.
    (tt (>name),'1') fappends fname
    (tt (>name),'2') fappends fname
    (tt (>name),'3') fappends fname
  end.
  (cctab codelist 0) fappends fname
  ('\end{document}', LF) fappends fname
  'done'
)
```

This script creates various data structures. For example the noun 'STAFFCOURSES' is shown in Figure 4.1.

Timetables for each lecturer, course, room or degree can be produced, and then formatted using LaTeX. In particular, 'alltt 0' produces the file \temp\alltt.tex containing timetables for each member of staff and for each degree course.



STAFFCOURSES

HPW	ST305
JBC	ST323 ST329 ST332
JEHS	seminar ST104 ST104p ST217/b ST217s/b ST329 ST952
JQS	ST114 ST301
JW	ST318
PJH	ST217/a ST217s/a ST304 ST329
RJR	ST113 ST215 ST327
SDJ	Stoch ST108 ST111/a ST111/b ST112/a ST112/b ST208 ST213
WSK	ST202 ST333

Figure 4.1: Noun `STAFFCOURSES` created by `timetab.js`

4.3.2 TEX Timetables Produced by J

An abbreviated listing of the output file `\temp\alltt.tex` follows; it can be processed by LaTeX 2_ε in the usual way [9, 18], finally producing tables like that shown in Figure 4.2.

```
% All timetables 11 Dec 97 18:05:32
% Produced using \j304\jehs\misc\timetab.js
%
\documentclass[a4paper]{report}
\pagestyle{empty}
%
\setlength{\textheight}{25cm}
\setlength{\textwidth}{18cm}
\setlength{\topmargin}{-1cm}
\setlength{\oddsidemargin}{-1cm}
\setlength{\evensidemargin}{-1cm}
\setlength{\tabcolsep}{4pt}  % default 6pt
%
\newcommand{\ttentry}[3]
  {\rule{0pt}{13pt}\hfil {\large #1}\hfil\break\null\hfil
  {#2}\hfil\break\null\hfil \raisebox{3pt}{\scriptsize #3}\hfil}
\newcommand{\ttnov}[1]{\multicolumn{1}{c}{#1}}
\newcommand{\ttstrut}{\rule[-28pt]{0pt}{41pt}}
\newcommand{\tttimes}{\ttnov{\rule[-7pt]{0pt}{1pt}} &
 \ttnov{ 9--10} &
 \ttnov{10--11} &
 \ttnov{11--12} &
 \ttnov{12--13} &
 \ttnov{13--14} &
 \ttnov{14--15} &
 \ttnov{15--16} &
 \ttnov{16--17} &
 \ttnov{17--18} }

%==========================================================================

\begin{document}
%
\begin{center}
 {\Large\bf MORSE Year 1 Term 1} \par \bigskip
\begin{tabular}{|c|*{9}{p{1.55cm}|}}
 \tttimes
\\ \hline
 \ttstrut Mon
 & &
 \ttentry{MA125}{GLT1}{w:6-10}
 &
```

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

```
\newpage

%==========================================================================

\begin{center}
 {\Large\bf Maths \& Stats Year 2 Term 1} \par \bigskip
\begin{tabular}{|c|*{9}{p{1.55cm}|}}
 \tttimes
\\ \hline
 \ttstrut Mon
 &
 \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}
 &
 \ttentry{\bf ST208}{F110}{w:2-10}
 &
 \ttentry{IB106}{R0.21}{w:1-6,8-10}
 & &
 \ttentry{MA231}{L3}{w:1-10}
 &
```

```
 \ttentry{\bf ST217}{PLT}{w:2-10}
 &
\\ \hline
 \ttstrut Tues
 & & &
 \ttentry{MA241}{H051}{w:1-10}
 &
 \ttentry{IB207}{S013}{w:1-5} \break
 \ttentry{IB207}{L4}{w:6-10}
 &
 \ttentry{MA242}{GLT1}{w:1-10}
 &
 \ttentry{\bf ST208}{rm68}{w:1-10}
 &
 \ttentry{\bf MA244}{GLT1}{w:1-10}
 &
 \ttentry{IB207}{S021}{w:1-10}
 &
\\ \hline
 \ttstrut Wed
 &
 \ttentry{\bf ST202}{ACCR}{w:1-10}
 &
 \ttentry{MA241}{L3}{w:1-10}
 &
 \ttentry{\bf ST217}{H052}{w:1-10}
 &
 \ttentry{IB215}{H051}{w:1-10} \break
 \ttentry{\bf MA244}{R0.21}{w:1-10}
 &
 \ttentry{EC213}{S021}{w:1-10}
 & & & &
\\ \hline
 \ttstrut Thurs
 &
 \ttentry{\bf ST217}{ACCR}{w:1-10}
 &
 \ttentry{EC213}{LIB2}{w:1-10}
 &
 \ttentry{MA231}{ACCR}{w:1-10}
 & &
 \ttentry{MA242}{GLT1}{w:1-10}
 &
 \ttentry{IB106}{R0.21}{w:1-6,8-10} \break
 \ttentry{\bf ST208}{F110}{w:1-10}
 &
 \ttentry{EC213}{L5}{w:1-10}
 &
 \ttentry{IB207}{L4}{w:1-10}
 &
\\ \hline
 \ttstrut Fri
 &
 \ttentry{\bf MA244}{L3}{w:1-10}
 &
 \ttentry{\bf ST202}{PLT}{w:1-10}
 & & &
 \ttentry{MA242}{L3}{w:1-10}
 &
 \ttentry{MA231}{R0.21}{w:1-10} \break
 \ttentry{\bf ST208}{F110}{w:1-10}
 & & &
\\ \hline
\end{tabular}
\end{center}
\newpage

%===========================================================================

\begin{center}
 {\Large\bf Maths \& Stats Year 2 Term 2} \par \bigskip
\begin{tabular}{|c|*{9}{p{1.55cm}|}}
 \tttimes
\\ \hline
 \ttstrut Mon
 &
```

```
 \ttentry{MA205}{R0.21}{w:1-10} \break
 \ttentry{PH201}{S020}{w:1-10}
 &
```

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

```
 \ttstrut Fri
 & & & & & & & & &
\\ \hline
\end{tabular}
\end{center}
\newpage

%=========================================================================

{\small
\begin{center}
{\Large\bf Course Codes} \par
\bigskip
\end{center}
\begin{tabular}{ll}
 CS117 & Programming for Scientists \\
 CS128 & Discrete Mathematics II \\
 EC106 & Introduction to Quantitative Economics \\
 EC208 & Economics of the Firm and Industry \\
```

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

```
 ST329 & Topics in Statistics \\
 ST332 & Medical Statistics \\
 ST333 & Applied Stochastic Processes \\
 ST952 & Introduction to Statistical Practice \\
 STsem & Statistics Departmental Seminar \\
 Stoch & Stochastic Calculus + Control\end{tabular}
%
} % end{\small}
\newpage%
\end{document}
```

Maths & Stats Year 2 Term 1

	9–10	10–11	11–12	12–13	13–14	14–15	15–16	16–17	17–18
Mon	PH201 S020 w:1-10	**ST202** ACCR w:2-10	IB215 ACCR w:1-10 MA241 L3 w:2-10	**ST208** F110 w:2-10	IB106 R0.21 w:1-6,8-10		MA231 L3 w:1-10	**ST217** PLT w:2-10	
Tues			MA241 H051 w:1-10	IB207 S013 w:1-5 IB207 L4 w:6-10	MA242 GLT1 w:1-10	**ST208** rm68 w:1-10	**MA244** GLT1 w:1-10	IB207 S021 w:1-10	
Wed	**ST202** ACCR w:1-10	MA241 L3 w:1-10	**ST217** H052 w:1-10	IB215 H051 w:1-10 **MA244** R0.21 w:1-10	EC213 S021 w:1-10				
Thurs	**ST217** ACCR w:1-10	EC213 LIB2 w:1-10	MA231 ACCR w:1-10		MA242 GLT1 w:1-10	IB106 R0.21 w:1-6,8-10 **ST208** F110 w:1-10	EC213 L5 w:1-10	IB207 L4 w:1-10	
Fri	**MA244** L3 w:1-10	**ST202** PLT w:1-10			MA242 L3 w:1-10	MA231 R0.21 w:1-10 **ST208** F110 w:1-10			

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

References

[1] F. J. Anscombe. *Computing in Statistical Science through APL.* Springer-Verlag, New York, 1981.

[2] C. Burke. *J Phrases.* Iverson Software Inc., `http://www.jsoftware.com/`, 1996.

[3] C. Burke. *J User Manual.* Iverson Software Inc., `http://www.jsoftware.com/`, 1996.

[4] C. Burke. APL and J. available at `http://www.jsoftware.com/`, 1997.

[5] C. Burke. Locales in J. available at `http://www.jsoftware.com/`, 1997.

[6] C. Burke and R. Hui. J for the APL programmer. available at `http://www.jsoftware.com/`, 1997.

[7] M. H. DeGroot. *Probability and Statistics.* Addison-Wesley, Reading, Mass., second edition, 1989.

[8] G. H. Golub and C. F. van Loan. *Matrix Computations.* The John Hopkins University Press, Baltimore, Maryland, second edition, 1989.

[9] M. Goossens, F. Mittelbach, and A. Samarin. *The LaTeX Companion.* Addison-Wesley, Reading, Mass., 1994.

[10] ISI. *J User Conference Proceedings: Proceedings of the first J User Conference in Toronto, Ontario, June 24–25, 1996.* Iverson Software Inc., `http://www.jsoftware.com/`, 1996.

[11] ISI. *J Release Notes.* Iverson Software Inc., `http://www.jsoftware.com/`, 1997.

[12] E. B. Iverson. *J Primer.* Iverson Software Inc., `http://www.jsoftware.com/`, 1996.

[13] K. E. Iverson. *A Dictionary of APL.* APL Press, `http://www.jsoftware.com/`, 1986.

[14] K. E. Iverson. *Concrete Math Companion.* Iverson Software Inc., `http://www.jsoftware.com/`, 1996.

[15] K. E. Iverson. *J Introduction and Dictionary.* Iverson Software Inc., `http://www.jsoftware.com/`, 1996.

[16] K. E. Iverson. *Exploring Math.* Iverson Software Inc., `http://www.jsoftware.com/`, 1997.

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 1988.

[18] L. Lamport. *LaTeX User's Guide & Reference Manual.* Addison-Wesley, Reading, Mass., 1986.

[19] C. A. Reiter. *Fractals, Visualization and J.* Iverson Software Inc., `http://www.jsoftware.com/`, 1995.

[20] K. Smillie. JSP: A J statistical package. Technical report, Department of Computing Science, University of Alberta, 1995. available from `ftp.cs.ualberta.ca`.

[21] K. Smillie. Some notes on introducing J with statistical examples. Technical report, Department of Computing Science, University of Alberta, 1995. available from `ftp.cs.ualberta.ca`.

[22] K. Smillie. A J implementation of resampling stats. *Gimme Arrays*, 3(7):7–18, 1996.

[23] K. Smillie. Teaching with J—an example from statistics. Technical report, Department of Computing Science, University of Alberta, 1996. available from `ftp.cs.ualberta.ca`.

[24] K. Smillie. Understanding data with J. Technical report, Department of Computing Science, University of Alberta, 1996. available from `ftp.cs.ualberta.ca`.

[25] A. Smith. Mostly on OpenGL. *Vector*, 14(2):50–54, 1998.

[26] A. Sykes and T. Stroud. APL and nested arrays—a dream for statistical computation. *Vector*, 14(2):58–70, 1998.