A Brief J Reference

This brief reference gives informal descriptions of most of the J primitives. Not every primitive is included and some idioms, examples and other resources have been added where appropriate. Since the presentation is brief and informal, it is not a replacement for the main J references: the *J Introduction and Dictionary*, the *J User manual* and the *J Primer*.

However, since the material is informally organized by topic, this reference may be useful when considering which J features might be relevant to a given problem. Some users may also find it helps locate gaps in knowledge that can then be filled in with the main references.

Chris Burke
Jsoftware Inc.
cburke@jsoftware.com
`www.jsoftware.com`

Cliff Reiter
Department of Mathematics
Lafayette College
`www.lafayette.edu/~reiterc`

Last updated April 2008 for J602.

Contents

1 Language Basics

Examples:

```
   fahrenheit =: 50
   (fahrenheit - 32) * 5%9
10

   prices=: 3 1 4 2
   orders=: 2 0 2 1
   orders * prices
6 0 8 2
   +/ orders * prices
16

   +/ \ 1 2 3 4 5
1 3 6 10 15
   2 3 * / 1 2 3 4 5
2 4 6  8 10
3 6 9 12 15

   cube=: ^&3
   cube i. 9
0 1 8 27 64 125 216 343 512
```

Names

`50 fahrenheit`	Nouns
`+ - * %`	Verbs
`/ \`	Adverbs
`&`	Conjunction
`( )`	Punctuation
`=:`	Assignment

- Verbs act upon nouns (their arguments) to produce noun results
- A verb may have two distinct (usually related) meanings depending on whether it is applied to one argument (to its right), or to two arguments (left and right).
- An adverb acts on a single noun or verb to its left, typically returning a verb. For example: +/ is a verb that sums its argument.
- A conjunction applies to two arguments, either nouns or verbs, typically returning a verb. In the example above, `^&3` is the verb *cube*.

2 Nouns

Nouns are classified in three independent ways:

- numeric or literal
- open or boxed
- arrays of various ranks

The atoms of any array must belong to a single class: numeric, literal, or boxed. Arrays of ranks 0, 1, and 2 are also called atom, list, and table, or, in mathematics, scalar, vector, and matrix.

A single entity such as `2.3` or `'A'` is called an atom. The number of atoms in the shape of a noun is called its rank. Each position of the shape is called an axis of the array, and axes are referred to by indices 0, 1, 2, etc.

Boxed nouns are produced by the verb `<` (box). The result of box is an atom, and boxed nouns are displayed in boxes. Box allows one to treat any array (such as the list of letters that represent a word) as a single entity, or atom.

3 Constants

`r`	rationals; `5r4` is $\frac{5}{4}$
`b`	base representations; `2b101` is 5
`e`	base 10 exponential notation (scientific notation); `1.2e14` is 1.2×10^{14}
`p`	base π exponential notation; `3p6` is $3\pi^6$
`x`	base e exponential notation; `3x2` is $3e^2$
`x`	extended precision; `2^100x` is the exact integer 2^{100}
`j`	complex numbers; `3j4` is $3 + 4i$
`ad`	angle in degrees; `1ad45` is approximately `0.707j0.707`
`ar`	angle in radians; `1ar1` is `^0j1`

`a.`	*alphabet*; list of all 256 ASCII characters
`a:`	*boxed empty*
`_1`	*negative one* (-1)
`_`	*infinity* (∞)
`__`	*negative infinity* $(-\infty)$
`_.`	*indeterminate*

4 Basic Arithmetic

`x + y`	`x` *plus* `y`
`+ y`	`y`; *identity* function for real `y`, *conjugate* for complex `y`
`x - y`	`x` *minus* `y`
`- y`	*negate* `y`
`x * y`	`x` *times* `y`
`* y`	*signum* of `y` is `_1`, `0` or `1` depending on the sign of real `y`
`x % y`	`x` *divide* `y`
`% y`	*reciprocal* of `y`
`+: y`	*double* `y`
`-: y`	*halve* `y`
`*: y`	*square* `y`
`x %: y`	`x`*th root* of `y`
`%: y`	*square root* of `y`
`x ^ y`	`x` to the *power* `y`
`^ y`	*exponential* base e
`x ^. y`	base `x` *logarithm* of `y`
`^. y`	*natural logarithm* (base e)
`x | y`	*residue* (remainder); `y` mod `x`
`| y`	*absolute value* of `y`
`x <. y`	*minimum of* `x` and `y`; (*smaller of*, *lesser of*)
`<. y`	greatest integer less than or equal to `y`; called the *floor*
`x >. y`	*maximum of* `x` and `y`; (*larger of*, *greater of*)
`>. y`	least integer greater than or equal to `y`; called the *ceiling*
`<: y`	*predecessor* of `y`; `y-1` (*decrement*)
`>: y`	*successor* of `y`; `y+1` (*increment*)

5 Boolean and Relational Verbs

Result of tests are `0` if false or `1` if true.

`x < y`	test if `x` is *less than* `y`
`x <: y`	test if `x` is *less than or equal* to `y`
`x = y`	test if `x` is *equal* to `y`
`x >: y`	test if `x` *greater than or equal* to `y` (*larger than or equal*)
`x > y`	test if `x` is *greater than* `y` (*larger than)*
`x ~: y`	test if `x` is *not equal* to `y`
`x -: y`	test if `x` is identically same as `y` (*match*)
`-. y`	*not* `y`; `1-y` for numeric `y`.
`x +. y`	`x` *or* `y`; the greatest common divisor (*gcd*) of `x` and y
`x *. y`	`x` *and* `y`; the least common multiple (*lcm*) of `x` and `y`
`x +: y`	`x` *nor* `y` (*not-or*)
`x *: y`	`x` *nand* `y` (*not-and*)
`x e. y`	test if `x` is an item *in* `y` (*member of*)
`e. y`	test if the *raze* is *in* each open
`x E. y`	mark beginnings of list `x` as a sublist in `y` (*pattern occurrence*)

Boolean tests are subject to a default comparison tolerance of `t=:2^_44`. For example, `x=y` is `1` if the magnitude of the difference between `x` and `y` is less than `t` times the larger of the absolute values of `x` and `y`. The comparison tolerance may be modified with the fit conjunction, `!.`, as in `x=!.0 y`, tests if `x` and `y` are the same to the last digit.

6 Name Assignment

`abc=: 1 2 3`	*global assignment* of `1 2 3` to name `abc`
`abc=. 1 2 3`	*local assignment* of `1 2 3` to name `abc`. The value is only available inside the definition where it is made.
`'abc'=: 1 2 3`	*indirect assignment* of `1 2 3` to name `abc`
`'a b c'=: 1 2 3`	*parallel assignment* of `1` to `a`, `2` to `b` and `3` to `c`.
`'a b'=: 1 2;3`	*parallel unboxed assignment* of `1 2` to `a` and `3` to `b`
`cube=: ^ & 3`	function assignment
`(exp)=: 1 2 3`	result of expression `exp` is assigned `1 2 3`
`names ''`	list of names defined in current locale
`erase 'a b c'`	erases the names `a`, `b`, and `c`
`(4!:5) 1`	(snap) returns names changed since last execution of `(4!:5) 1`

Several foreign conjunctions of the form `4!:n` deal with names. See also the Locales lab to learn about using locales to create different locations for global names.

7 Data Information and Building

`# y`	*number* of items in `y` (*tally*)
`$ y`	*shape* of array `y`
`x $ y`	shape `x` *reshape* of `y` (cyclically using/reusing data)
`i. y`	list of *indices* filling an array of shape `y` (*integer*); negative `y` reverses axis
`i: y`	*symmetric arithmetic sequence*; for integer `y`, the integers from `-y` to `y`
`i: a j. b`	list of numbers from `-a` to `a` in `b` equal steps
`x F/ y`	*table* of values of dyad `F` with arguments from `x` and `y` (*outer product*)
`x , y`	*append* `x` to `y` where axis 0 is lengthened (*catenate*)
`x ,. y`	*stitch* `x` beside `y` (append items) where axis `1` is lengthened;
`x ,: y`	x *laminated* to `y` giving an array with 2 items
`, y`	*ravel* (string out) elements of `y`
`,. y`	*ravel items* of `y`
`,: y`	*itemize*, make `y` into a single item by adding a new length-1 leading axis
`$. y`	*sparse* matrix representation of `y`

8 Data Copying

`x # y`	*replicate* or *copy* items in y the number of times indicated by `x`; the imaginary part of `x` is used to specify the size of expansion with fill elements
`x #^:_1 y`	*expand* (inverse of `#`) selects items of `y` according to `1` in Boolean `x`, pads with fills where `0` in `x`
`(G # ])y`	selects elements of `y` according to Boolean test `G`; thus, `(2&< # ]) y` gives the elements of `y` greater than 2.
`x {. y`	shape `x` *take* of `y`; negative entries cause take from end of axes; entries larger than axis length cause padding with fill elements.
`{. y`	item in `1 {. y` for non-empty arrays; in general `0{y` (called *head*)
`{: y`	item in `_1{.y` or `_1{y` (called *tail*)
`x }. y`	*drop* shape `x` part of `y`; negative entries cause drop from end of axes.
`}. y`	`1 }. y` (one drop) or *behead*
`}: y`	`_1 }. y` (negative one drop) or *curtail*
`|: y`	transpose of y
`(<0 1) |: y`	trace (diagonal) of matrix `y`

9 Data Indexing and Amendment

`I { y`	item at position I in y (*index* or *from*) Arrays of I give corresponding arrays of items. Boxed arrays I give positions on corresponding axes. An empty box gives all values along that axis.
`x I } y`	y *amended* at positions I by data x

```
   [A=: i.3 4
0 1 2 3
4 5 6 7
8 9 10 11

   0 2 { A
0 1 2 3
8 9 10 11

   0 2 {"1 A
0 2
4 6
8 10

   (<0 2;3) { A
3 11

   1000 (<0 2;3) } A
0 1 2 1000
4 5 6 7
8 9 10 1000
```

10 Boxed Arrays

`< y`	*box* `y`
`> y`	*open* (unbox) `y` one level
`x ; y`	*link* `x` and `y`; box `x` and append to `y`; if `y` is unboxed, then box `y` first
`; y`	*raze* `y`; remove one level of boxing
`F &.> y`	apply `F` inside of each boxed element of `y`
`F &> y`	apply `F` to inside of each boxed element of `y` and adjoin the results.
`a:`	*boxed empty* (noun called *ace*)
`;: y`	boxed list of J words in string `y`; (*word formation*)
`L. y`	*depth* or deepest *level* of boxing in y
`F L: n y`	apply `F` at *level* `n` and maintain boxing. May be used dyadically and left and right level specified. If boxing is thought of as creating a tree structure, then adverb `L: 0` may be called *leaf*
`F S: n y`	apply `F` at level `n` and list the result (*spread*)
`{:: y`	*map* has the same boxing as `y` and gives the paths to each leaf
`x {:: y`	*fetch* the data from `y` specified by the path `x`

11 Rank

Rank can be specified by one, two or three numbers. If the rank `r` contains three numbers, the first is the monadic rank, the second the left dyadic rank and last the right dyadic rank. If it contains two numbers, the first gives the left dyadic rank and the second gives the monadic and right dyadic rank. All the ranks are the same when a single number is given.

`F"r y`	apply `F` on rank `r` cells of the data `y`
`x F"(lr,rr) y`	apply `F` on rank `lr` cells from `x` and rank `rr` cells from y
`x F"0 _ y`	table builder when `F` is a scalar function
`N"r`	constant function of rank `r` and result `N`
`F b. 0`	gives the monadic, left and right ranks of the verb `F`

12 Explicit Definition

Explicit definitions can be made with `m : n` where `m` is a number that specifies whether the result is a noun, verb, adverb or conjunction. When `n` is 0, successive lines of input give the defining expressions until an isolated, closing right parenthesis is reached. Noun arguments to adverbs and conjunctions may be specified by `m` on the left and `n` on the right. Verb arguments are `u` and `v` and the derived functions use `x` and `y` to denote their arguments.

`4 : 0`	input mode for a dyadic verb
`3 : 0`	input mode for general verb. This is the monadic definition, optionally followed by an isolated colon and the dyadic definition.
`2 : 0`	input mode for conjunction
`1 : 0`	input mode for an adverb
`0 : 0`	input mode for a noun

The right argument `n` as in `m : n`, may be a string, a `CRLF` delimited string, a matrix, or a boxed list of strings. For example:

`f=: 3 : '(*:y) + ^y'` defines $f(y) = y^2 + e^y$

`13 : n` converts to tacit form of a verb (if possible). For example:

```
   13 : 'x , 2 * x + y'
[ , 2 * +
```

13 Tacit Definition

In a tacit definition the arguments are not named and do not appear in the definition.

In many cases the tacit form of definition is much simpler and more obvious than the equivalent explicit definition.

For example:

`plus=: +`	assigns name `plus` to `+`
`sum=: +/`	sum of numeric list
`max=: >./`	maximum of numeric list
`mean=: +/ % #`	average of numeric list

Compare the last definition with an equivalent explicit definition:

```
mean=: 3 : '(+/y) % #y'
```

14 Verb Application to Subsets

`F/ y`	*insert* verb F between items of y, also called F-reduction; thus `+/2 3 4` is `2+3+4`
`G\ y`	apply `G` to *prefixes* of `y`; generalized scan
`F/ \ y`	`F` *scan* of `y`
`x G\ y`	apply `G` to sublists of length `x` in `y` (the lists are *infixes*); negative `x` gives non-overlapping sublists. For example `x avg y` gives length `x` *moving averages* of data in `y` (where `avg=: +/ % #`).
`G\. y`	apply `G` to *suffixes* of `y` (order of execution makes this fast!)
`x G\. y`	apply `G` to lists where sublists of length `x` in `y` are excluded (the sublists are *outfixes*)
`x G;._3 y`	*cut*; apply `G` to shape `x` tessellations of `y`. In general, the rows of `x` give the shape and offset used for the tessellation. Include shards by specifying `3` instead of `_3`.
`G;.3 y`	*cut*; generalized suffix
`G;._2 y`	*cut*; apply `G` to sublists marked by ending with the last item in y. So `}:<@}:;._2 y,CRLF` gives the boxed lines of `CRLF` delimited text `y`. `G;.2 y` includes marked positions in sublists. `G;._1 y` and `G;.1 y` use first item to mark beginnings of sublists. `G;.0 y` and dyads and gerunds `G` are also defined.
`x F/. y`	function `F` is applied to parts of `x` selected by distinct items (*keys*) in `y`. For example `#/.~ y` gives *frequency* of occurrence of items of y
`F/. y`	apply `F` to *oblique* lists from `y`.

15 Gerunds and Controlled Application of Verbs

`^:`	*iterate function* (*power*)
`F^:n y`	iterate `F` `n` times on `y`; see the Dictionary for gerund `n`
`F^:_ y`	iterate `F` until convergence (*limit*)
`F^:(i.n)y`	result of `F` iterated `0` to `n-1` times on `y`
`F^:G^:_ y`	iterate `F` on `y` until `G` gives false
`F`G`	*tie* verbs `F` and `G` together forming a gerund
`F/. y`	evaluate each verb in gerund `F` taken cyclically on data `y` (*evoke gerund*)
`F`:0 y`	alternative form of *evoke gerund*, returning all combinations of functions from `F` on y
`F@.G`	*agenda*: use `G` to select verb from gerund `F` to apply
`F::G`	*adverse*: apply `F`, if an error occurs, apply `G` instead

Many adverbs and conjunctions have gerund meanings that give generalizations; e.g. gerund insert cyclically inserts verbs from the gerund. Thus, the following are the same:

```
+`% / 1 2 3 4
1 + 2 % 3 + 4
```

16 Program Flow Control

Execution control is provided by words such as: `if. else. while.` etc. These control words:

- can occur anywhere in a line of code
- group the code into blocks

Here, a block is zero or more sentences, which may themselves contain control words. A J block is true if the first element of the result is not zero. In particular, an empty block is true.

In all cases, the result of the last expression executed that was not a test, is returned as the verb result.

if. elseif.

```
signum=: 3 : 0
if. y < 0 do. _1
elseif. y=0 do. 0
elseif. do. 1
end.
)

   signum &> _5 7 8 0
_1 1 1 0
```

Compare:

```
   * _5 7 8 0
_1 1 1 0
```

while. whilst.

The control word `while.` executes the loop while the control condition is true.

`whilst.` is the same as `while`, except the steps of the loop are executed once before the control condition is tested.

```
sumint=: 3 : 0
k=.0
s=.0
while. k<:y do.
  s=.s+k
  k=.k+1
end.
s
)

   sumint 10
55
```

for_name.

```
Sumint=: 3 : 0
s=.0
for_k. 1+i.y
   do. s=.s+k
end.
s
)

   Sumint 10
55
```

break.

The control word `break.` is used to step out of a `while.` or `whilst.` or `for_name.` loop, and `continue.` returns to the top of the loop. The control word `return.` can be used to exit from function execution.

select.

The `select.` control word allows execution of expressions when a value matches those in a given case or cases. An empty case matches all.

```
atype=: 3 : 0
select. #$y
case. 0 do. 'scalar'
case. 1 do. 'vector'
case. do. 'array of dimension greater than 1'
end.
)
```

```
   atype <i.3 3
scalar

   atype 'abc'
vector

   atype i.3 3 3
array of dimension greater than 1
```

try. catch.

The following line runs `expression2` if running `expression1` causes an error.

```
   try. expression1 catch. expression2 end.
```

There are also control words for labeling lines and going to those lines: `label_name.` and `goto_name.` .

17 Recursion

One can use self reference of verbs that are named. For example, the factorial can be computed recursively as follows.

```
   fac=: 3 : 'if. y <: 1 do. 1 else. y * fac y - 1 end.'
   fac 3
6
```

Also:

```
   fac=: 1:`(*fac@<:)@.*
   fac 3
6
   fac"0 i. 6
1 1 2 6 24 120
```

One can also create a recursive function without naming the function by using `$:` for self-reference. The factorial function can be defined recursively without name as follows.

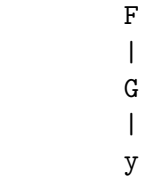
```
   (1:`(*$:@<:)@.*) 3
6

   (1:`(*$:@<:)@.*)"0 i.6
1 1 2 6 24 120
```

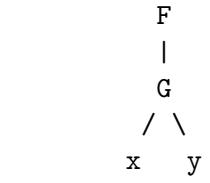
18 Function Composition

Atop

```
F@G y
```

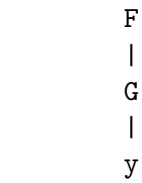



```
x F@G y
```

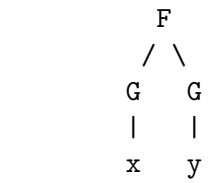


Compose

```
F&G y
```

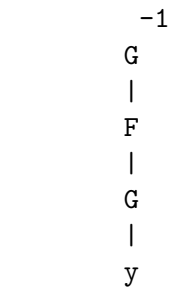



```
x F&G y
```

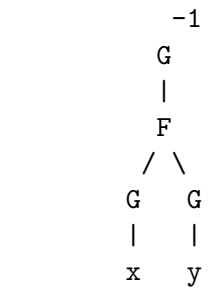


Under

```
F&.G y
```

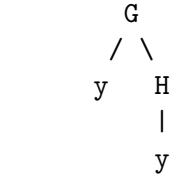



```
x F&.G y
```

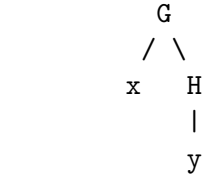


Hook

```
(G H) y
```

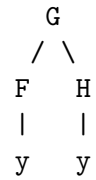



```
x (G H) y
```

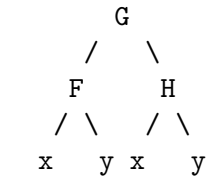


Fork

```
(F G H) y
```




```
x (F G H) y
```



The rank of `F@G` and `F&G` is the rank of `G`. *At* is denoted `@:` and is the same as `@` except the rank is infinite. *Appose* is denoted `&:` which is the same as `&` except the rank is infinite. The ranks of the hook and fork are infinite.

Longer trains of verbs are interpreted by taking forks on the right. Thus `F G H J` is the hook `F (G H J)` where `G H J` is a fork, and `F G H J K` is the fork `F G (H J K)`.

Under

The verb `u &. v` is equivalent to the composition `u & v` except that the verb obverse (inverse) to `v` is applied to the result for each cell. For example, multiplication is sum under log:

```
   3 + &. ^. 4
12
```

However, the rank of the result of `u &. v` is the monadic rank of `v`, which for many verbs is zero, whereas it is often the case that you want the rank to be infinite. An alternate form of under is `&.:`, which is equivalent to `u &. (v"_)`. For example:

```
   +/ &. ^. 3 4 5
3 4 5
   +/ &.: ^. 3 4 5
60
```

Cap

`[: F G` has the effect of passing no left argument to `F` as part of the fork - the left branch of the fork is capped - thus `F` is applied monadically.

19 More Verbs from Verbs

`N&G`	monad derived from dyad `G` with `N` as the fixed left argument
`G&N`	monad derived from dyad `G` with `N` as the fixed right argument (known as *bond* or *curry*)
`F~y`	reflects `y` to both arguments; i.e. `y F y` (*reflex*)
`x F~ y`	*pass* interchanges arguments; i.e. `y F x` (*commute*)
`F : G`	function with monad `F` and dyad `G` (*monad/dyad* definition)
`F :. G`	function `F` with *obverse* (restricted inverse) `G`
`G f.`	function `G` with names appearing in its definition recursively replaced by their meaning. This *fix*es (makes permanent) the function meaning
`F b. _1`	*obverse* (*inverse*) of `F`
`F b. 1`	*identity function* for `F`

20 Conversion: Literal, Numeric, Base, Binary

`": y`	*format* array `y` as a literal array
`a.b ": y`	*format* data in `y` with field width a and b decimal digits
`ajb ": y`	*format* data in `y` with field width a and b decimal digits, for example: `15j10 ": o.i.3 4`
`":!. c y`	*format* data showing `c` significant digits
`". y`	*execute* or *do* string `y`
`x ". y`	convert `y` to numeric using `x` for illegal numbers. J syntax is relaxed so appearances of `-` in `y` are treated like `_`
`".@}: ;._2 y`	*execute* expressions in `CRLF` delimited substrings appearing in y (that ends with `CRLF`) and adjoin the results
`'m'~`	value of name `m` is *evoked*
`#: y`	binary representation of `y` (*antibase-two*)
`x #: y`	representation of `y` in base `x` (*antibase*)
`#. y`	value of binary rank-1 cells of `y` (*base-two*)
`x #. y`	value of base `x` rank-1 cells of `y` (*base*)
`3 !: n`	various binary conversions; for example, `1 (3 !:4) y` converts J floats to binary short floats while `_1 (3!:4) y` converts binary short floats to J floats.
`8 !: n`	format `y` according to format phrase `x`. For example, format to width 11, decimal places 2, comma-separated, with zeros formatted to `nil`, and infinities to `n/a`:

```
   'b<nil>d<n/a>c11.2' (8!:2) 1.23 12345 0.123,__ 0 _1234.5,:_44 0.5 0.1
      1.23  12,345.00       0.12
       n/a         nil  -1,234.50
    -44.00        0.50       0.10
```

See also the Foreign Conjunction help.

21 Reading and Writing Files

The following verbs are based on foreign conjunctions in the form `1!:n`. These provide for file reading/writing including indexed reads and writes and creating directories, reading and setting attributes and permissions. Chopping file data in appropriate places can be accomplished with `;._2` (cut). Simple substitution (e.g., `_` for `-`) may be accomplished with `charsub` from `strings`. See `regex` for more complex processing. Memory mapped files should be considered for huge data sets.

`1!:0 y`	directory information matching path and pattern in `y` (see `fdir`)
`1!:1 y`	read file `y` specified by a boxed name (see `fread` and `freads`)
`x 1!:2 y`	write file `y` with raw, (`a.` or text) data `x` (see `fwrite` and `fwrites`)
`x 1!:3 y`	append file `y` with raw, `a.` or text data `x` (see `fappend` and `fappends`)
`1!:11 y`	indexed read. `y` is a pair: file name; index and length. The index may be negative. If the length is elided, the read goes to the end.
`x 1!:12 y`	indexed write. `y` is a pair: file name; index.

For example:

```
   'abcdefgh' 1!:2 <F=: 't1.dat'
   1!:1 <F
abcdefgh
   1!:11 F;2 5
cdefg
   'XYZ' 1!:12 F;3
   1!:11 F;2 5
cXYZg
```

Files may also be referenced by number; keyboard and screen input/output are supported, and other facilities give other useful file access including indexed i/o, permissions, erasure, locking, attributes.

Convenient utilities are defined in the `files` library script. For example, read in a file, returning the result in a matrix:

```
   load 'files'
   'm' fread 'mydata.dat'
```

See also the *Files* lab.

22 Scripts

Scripts are plain text files containing J expressions. Typically the file extension is `.ijs`. Loading the scripts runs the J expressions.

`Ctrl+N`	open a new script window
`0!:0 <'filename.ijs'`	run the script `filename.ijs`; note boxing of filename
`0!:1 <'filename.ijs'`	run the script with display
`0!:0 y`	run the J noun `y` as a script
`0!:1 y`	run the J noun `y` displaying the result
`0!:10 y`	run the J noun `y` and continue on errors
`load 'filename.ijs'`	similar to `0!:0` except that the script is loaded within an explicit definitions, and hence local definitions in the script do not exist upon completion. This allows a script to have definitions that are local to the script.
`load 'scriptname'`	loads a library script, for example *dates* is the script `system\main\dates.ijs`.
`require 'filename'`	similar to `load` except that if the script has already been loaded, it is not loaded again.

Typically, applications are built from several scripts.

Project Manager helps you manage your J script files. In particular, it lets build you applications from several scripts.

Scripts are maintained individually during development, and can be compiled into a single output script for distribution/runtime/installation purposes. You can customize the build to suit your application.

Run the Project Manager from menu Run—Project Manager... or by pressing Ctrl-B.

For more information, see the lab *Building Applications*.

23 Sorting and Searching

`/: y`	*grade up*; indices of items of `y` ordered so that the corresponding items of `y` would be in nondecreasing order
`x /: y`	*sort* `x` according to indices in `/:y`
`/:~ y`	*sorts* items of `y` into nondecreasing order
`/:/: y`	*rank order* of items in `y`
`\: y`	*grade down*; indices of items of `y` ordered so that the corresponding items of `y` are in nonincreasing order
`x i. y`	*indices* of items of `y` in the reference list `x`
`x e. y`	test if `x` is an item *in* `y` (*member of*)
`e. y`	test if the *raze* is *in* each open
`x E. y`	mark beginnings of list `x` as a sublist in `y` (*pattern occurrence*)
`I. y`	*indices* of 1 in boolean list `y`; thus `I.y<4` gives indices where y is less than 4
`x I. y`	*indices* of `y` in the intervals defined by `x`
`~. y`	*nub* of `y`; that is, items of `y` with duplicates removed (unique)
`({.,#)/.~y`	may use key to get *nub* and *frequencies* appearing in `y`
`~: y`	*nubsieve*: Boolean vector `v` so `v#y` is `~.y`
`= y`	*self-classify* `y` according to `~. y`
`(G # ])y`	selects items of y according to Boolean test `G`; thus, `(2&< # ])y` gives items of y greater than 2.
`x -. y`	items of `x` less those in `y`

24 Efficiency, Error Trapping, and Debugging

`6!:2 y`	*time* (seconds) required to execute string y. Optional left argument specifies number of repetitions used to obtain average run time
`7!:2 y`	*space* (bytes) required to execute string y
`u :: v`	result of applying verb u unless that results in an error in which case v is applied (*adverse*)
`try. e1 catch. e2 end.`	is similar except expressions in explicit definition mode are executed instead of verbs being applied

The *try/catch* control structure may contain one or more distinct occurrences of `catch. catchd. catcht.` (in any order). For example:

```
try. B0 catch. B1 end.
try. B0 catcht. B1 catchd. B2 end.
try. B0 catcht. B1 catch. B2 catchd. B3 end.
```

The `B0` block is executed and:

`catch.`	catches errors, whatever the setting of the debug flag `13!:0`
`catchd.`	catches errors, but only if the debug flag is `0`
`catcht.`	catches a `throw.` expression

The foreign conjunctions `13!:n` provide the underlying debugging facilitiesw, while the Debug application provides interactive debugging, see the Debug lab.

The Performance Monitor provides detailed execution time and space used when running an application, see the Performance Monitor lab.

25 Randomization and Simulation

`? y`	*random* index from `i.y`; called *roll*; for example, `+/\(?100#2){_1 1` is a 100 step random `_1 1` walk
`?. y`	*default random* index from `i.y` using 16807 as the random seed
`x ? y`	`x` random indices *dealt* from `i.y` without duplication
`? 0`	random number in range `[0,1)`
`9!:0 ''`	query random seed
`9!:1 y`	set random seed to `y`
`randomize ''`	randomize random seed; `randomize` is defined in *numeric.ijs*

See also `system\packages\stat\random.ijs` for various random number utilities, and `system\packages\stat\statdist.ijs` for utilities for selecting from various distributions.

For example:

```
   3 deal ;: 'anne henry mary susan tom'
+-----+----+---+
|susan|mary|tom|
+-----+----+---+

   normalrand 5  NB. mean 0, sd 1
0.719033 _0.512529 0.801304 0.436659 _0.0496758
```

26 Constant and Identity Verbs

`] y`	result is `y`, the *identity* function on `y`
`x ] y`	result is `y` (*right*)
`[ y`	result is `y`, the *identity* function on `y`
`x [ y`	result is `x` (*left*)
`+ y`	result is `y` if `y` is a real number
`0: y`	result is the scalar `0`
`1: y`	result is `1`; likewise, there are constant functions `_9:` to `9:`
`_: y`	result is the infinite scalar `_`
`N"r`	constant function with value `N` for each rank `r` cell

27 Exact Computations

`2x or 2r1`	exact integer 2 (*extended precision*)
`2x^100`	exact integer 2^{100}
`2r3`	exact rational number $\frac{2}{3}$ (*extended precision*)
`x: y`	convert `y` to extended precision rational
`x:^:_1 y`	convert `y` to fixed precision numeric
`2 x: y`	numerator and denominator of extended precision rationals

There is special code to avoid exponentiation for extended precision arguments when using residue, for example:

`m&|@(2x&^) y`	computes 2^y mod m efficiently (without computing 2^y)

28 Number Theory and Combinatorics

`p: y`	y-th *prime* number (in origin 0)
`p:^:_1 y`	number of primes less than `y`
`x p: y`	various number theoretic functions: next prime, totient, etc.
`q: y`	prime *factors* of `y`
`x q: y`	prime *factors* of `y` with limited factor base
`x +. y`	*greatest common divisor* (gcd)
`x *. y`	*least common multiple* (lcm)
`gcd y`	function `gcd` defined in `system\packages\math\gcd.ijs` results in the `gcd` of the elements of `y` along with the coefficients whose dot product with `y` gives the `gcd`. Also useful for finding inverses modulo *m*.
`x | y`	*residue* (remainder) `y` modulo `x`
`! y`	*factorial* of `y` for integer `y` and $\Gamma(y+1)$ in general
`x ! y`	number of *combinations* of `x` things from `y` things (generalized)
`A. y`	*atomic* representation (position) of permutation `y`
`x A. y`	applies permutation with atomic representation `x` to `y` (*atomic permute*, *anagram*). For example, `(i.!n) A. i.n` is all permutations of order `n`
`C. y`	*cycle* representation of numeric permutation `y` as a boxed list; visa versa when y is boxed
`x C. y`	*permutes* y according to permutation `x` (either in numeric or boxed cyclic representation)
`{ y`	*Cartesian* product: all selections of one item from each box in `y`.

29 Circular and Numeric Verbs

Many trigonometric functions and other functions associated with circles are obtained using o. with various numeric left arguments.

`o. y` πy (*pi times*)

`0 o. y`	$\sqrt{1-y^2}$ (*circle* functions)		
`1 o. y`	$\sin(y)$	`_1 o. y`	$\sin^{-1}(y)$
`2 o. y`	$\cos(y)$	`_2 o. y`	$\cos^{-1}(y)$
`3 o. y`	$\tan(y)$	`_3 o. y`	$\tan^{-1}(y)$
`4 o. y`	$\sqrt{1+y^2}$	`_4 o. y`	$\sqrt{y^2-1}$
`5 o. y`	$\sinh(y)$	`_5 o. y`	$\sinh^{-1}(y)$
`6 o. y`	$\cosh(y)$	`_6 o. y`	$\cosh^{-1}(y)$
`7 o. y`	$\tanh(y)$	`_7 o. y`	$\tanh^{-1}(y)$
`8 o. y`	$\sqrt{-(1+y^2)}$	`_8 o. y`	$-\sqrt{-(1+y^2)}$
`9 o. y`	real part(y)	`_9 o. y`	y
`10 o. y`	abs(y) which is |y	`_10 o. y`	conjugate(y)
`11 o. y`	imaginary part(y)	`_11 o. y`	yi where i is $\sqrt{-1}$
`12 o. y`	arg(y)	`_12 o. y`	e^{iy}

`m H. n y` *hypergeometric* function; sometimes denoted $F(m;n,y)$
`x m H. n y` hypergeometric function using `x` terms

30 Complex Numbers

Complex numbers are denoted with a `j` separating the real and imaginary parts. Thus, the complex number commonly written `3.1 + 4`i is denoted `3.1j4`.

`+ y` *complex conjugate* of `y`
`| y` *magnitude* of `y`
`* y` *generalized signum*; complex number in `y` *direction*
`j. y` complex number `0jy`; that is, iy (*imaginary*)
`x j. y` complex number `xjy`; that is, $x+iy$ (*complex*)
`+. y` pair containing real(y) and imaginary(y)
`*. y` polar pair (r,θ) where $y = re^{i\theta}$, (*length, angle*)
`r. y` is e^{iy} (*angle to complex*)
`x r. y` is xe^{iy} (*polar to complex*)

31 Matrix Arithmetic

`x +/ . * y`	*matrix product* of `x` and `y` (*dot product* for vectors)
`x +/ . = y`	number of places where vector arguments match
`x F/ . G y`	*inner product*; F-insert applied to pairwise `G`'s applied row by column; the last axis of `x` and first axis of `y` need to be compatible (same or 1) and that axis collapses in the product.
`x H . G y`	*inner product*; `H` applied to cells of `G` applied rank `_1 _`
`-/ . * y`	*determinant* of `y`
`F . G y`	*generalized determinant*; `+/ . *` gives the *permanent*.
`x %. y`	*matrix divide*; solution `z` to the linear matrix system `x = y +/ . * z`; least squares solution is given when appropriate
`%. y`	*matrix inverse* or *pseudo-inverse* of matrix `y`
`|: y`	*transpose* of `y`
`x |: y`	*generalized transpose* of `y`. Axes listed in `x` are successively moved to the end.
`|. y`	*reverse* items in `y`
`x |. y`	*rotate* items in `y` by `x` positions downward along the last axis
`=@i. y`	`y` by `y` identity matrix; multiply by diagonal to get a diagonal matrix
`128!:0 y`	*QR decomposition* of `y`

The J Addons *Lapack* and *FFTW* give extensive linear algebra and fast Fourier transform utilities, respectively.

32 Calculus, Roots and Polynomials

`F D. n y`	`n`-th derivative of `F` at `y`
`F d. n y`	`n`-th derivative rank zero: compare to `(F D. 1)"0 y`
`x F D: n y`	slope of secant of `F` at `y` and `x+y`
`F t. n`	`n`-th Taylor series coefficient of `F` about `0`
`F t: n`	`n`-th Taylor series coefficient of `F` about `0` weighted by `!n`
`F T. n y`	`n`-th degree Taylor polynomial for `F` about `0` evaluated at `y`
`p. y`	*polynomial*; toggles between coefficient representation and leading-coefficient-with-root boxed representation of polynomials.
`x p. y`	polynomial specified by `x` evaluated at points `y`. The coefficients `x` are in ascending powers, for example `2 1 3 p.` is the polynomial $3x^2 + x + 2$.
`p.. y`	coefficients of derivative of polynomial `y`
`x p.. y`	integral of polynomial `y` with a constant term `x`

33 Special Datatypes

Sparse arrays provide a compact and efficient storage form for very large arrays where most elements are zero or some other *sparse element*.

The verb `$.` converts a dense array to sparse, and `$.^:_1 y` (`$. inverse`) converts a sparse array to dense.

A sparse array has a single sparse element, plus an array of other values and a matrix of their corresponding indices.

The sparse attribute can be assigned to axes individually. Non-sparse axes are known as dense axes.

J primitives work directly on sparse arrays, and operations give the same results when applied to dense and sparse versions of the same arrays. In other words, the following identities hold for any function `f` , with the exception only of those (like overtake `{.`) which use the sparse element as the fill.

```
f -: f &. $.
f -: f &. ($.^:_1)
```

All primitives accept sparse or dense arrays as arguments (e.g. `sparse+dense` or `sparse$sparse`).

Symbols are a mechanism for searching, sorting, and comparisons on data that is much more efficient than alternatives such as boxed strings. Structural, selection, and relational verbs work on symbols.

The monad verb `s:` converts arrays into symbols. Several types of arguments are acceptable:

- string with the leading character as the separator
- literal array where each row, excluding trailing blanks, is the name of a symbol
- array of boxed strings

Unicode is a 2-byte (16-bit) character datatype.

The verb `u:` creates unicode arrays. The monad applies as follows:

Argument	**Result**
1-byte characters	same as `2&u:`
2-byte characters	copy of argument
integers	same as `4&u:`

The inverse of the monad `u:` is `3&u:`

The dyad `u:` takes a scalar integer left argument and applies to several kinds of arguments:

Left	Right	Result
`1`	2-byte characters	1-byte characters; high order bytes are discarded
`2`	1-byte characters	2-byte characters; high order bytes are `0`
`3`	2-byte characters	integers
`4`	integers	2-byte characters; integers must be from 0 to 65535
`5`	2-byte characters	1-byte characters; high order bytes must be `0` (and are discarded)
`6`	1-byte characters	2-byte characters; pairs of 1-byte characters are converted to 2-byte characters

`1&u:` and `2&u:` is an inverse pair, as are `3&u:` and `4&u:` .

34 Graphics

J offers a great number of facilities for doing Windows graphics. Running the Graphics, Open GL and Plot labs is recommended. The `plot.ijs` script provides a powerful high level set of useful utilities. Most users will do well to study the plot lab first. The scripts `gl2.ijs` and `gl3.ijs` provide the graphics functions for windows driver and opengl graphics functions.

The `opengl` package provides support for OpenGL graphics

The `image3` addon supports reading and writing various image formats and creating and organizing html galleries.

The `fvj3` addon provides materials for Cliff Reiter's book *Fractals, Visualization and J, 3rd edition.*

35 Session Manager Short-Cut Keys

Many J short-cut keys are defined and users may define their own. A few follow:

`Enter`	captures current line for editing on the execution input line
`F1`	help
`Ctrl+F1`	context sensitive help
`Ctrl+Shift-up-arrow`	scroll up in execution log history
`Ctrl+D`	window with execution history
`Ctrl+E`	load selection
`Ctrl+Shift+E`	load selection showing display
`Ctrl+Shift+1`	set mark 1 on current line (likewise 2-9)
`Alt+1`	go to mark 1 in current window (likewise 2-9)

See also menus `Edit|Configure|Fkeys` and `Edit|Configure|Shortcuts`. These are part of the system configuration in menu *Edit—Configure...*

36 Addons

There are several addon packages available from the J wiki, see http://www.jsoftware.com/jwiki/JAL, such as:

`fftw`	fast fourier transform package.
`image3, platimg`	facilities for reading and writing images in a variety of formats.
`lapack`	standard linear algebra package.
`publish`	builds pdf reports from markup.
`SFL`	The SFL (Standard Function Library) from iMatix is a portable function library for C/C++ programs.
`sax`	proXML parser based on Expat library
`SQLite`	provides J bindings to SQLite embedded engine
`tara`	reads and writes files in Excel format.

Install from menu `Run|Package Manager`.

37 Parts of Speech and Grammar

Most words are denoted with an ASCII symbol found on standard keyboards, or such a symbol followed by a period or colon. For example, we may think of `%` as denoting a J word meaning *reciprocal*, and `%.` as an inflection of that word meaning *matrix inverse*.

Basic data objects in the language are nouns. These include scalars, such as `3.14`, as well as lists (vectors) such as `2 3 5 7`, matrices which are a rectangular arrangement of atoms and higher dimensional arrays of atoms. In general, arrays contain atoms that are organized along axes. These arrays may be literal, numeric or boxed. Any array may be boxed and, thereby, be declared to be a scalar. Nested boxing allows for rich data structures.

The number of axes of an array gives its dimension. Thus, a scalar is 0-dimensional, a vector is 1-dimensional, a matrix is 2-dimensional and so on. The shape of an array is a list of the lengths of its axes. Often, the shape can be imagined as being split into two portions, giving an array of arrays. The leading portion of the split gives the frame (the shape of the outer array) and the other portion corresponds to the shape of the arrays, giving what are called *cells*. The items are the cells that occur by thinking of an n-dimensional array as a list of (`n-1`)-dimensional arrays. That is, items are rank `_1` cells.

Functions are known as verbs. For example, `+` denotes plus, `%:` denotes root, and (`+/ % #`) denotes average. Adverbs take one argument (often a verb) and typically result in a verb. For example, insert, denoted by `/` is an adverb. It takes a verb argument such as `+` and results in a derived verb `+/` that sums items. Notice that adverbs take arguments on the left. The derived verb may itself take one noun argument (where it is a monad) or two noun arguments (where it is a dyad). It is sometimes helpful to be able to view a function as an object that can be formally manipulated. This facility is inherent in the J gerund. Gerunds are verbs playing the role of a noun.

Conjunctions take two arguments and typically result in a verb. For example, `.` (dot) is a conjunction (be careful to distinguish this from a dot that is the last character in a name). For example, with left argument sum and right argument times, we get the matrix product `+/ . *` as the derived verb.

The application of verbs to arguments proceeds from right to left. Thus `3*5+2` is `21` since the `5+2` is evaluated first. However, it is possible to think of the expression as being read left to right: 3 times the result of `5` plus `2`. Therefore, verbs have long right scope and short left scope. Of course, one can use parentheses to order computations however desired: `(3*5)+2` is `17`.

In contrast to verbs, adverbs and conjunctions bond to their arguments before verbs do. Also in contrast, they have long left scope and short right scope. Thus, we do not need the parentheses in `(+/) . *` to denote the matrix product since the left argument of the dot is the entire expression on its left, namely, `+/` which gives the sum. Thus `+/ . *` denotes the matrix product.

38 Glossary

Adverb	A part of speech that takes an argument on the left and typically results in a verb. For example, insert `/` is an adverb such that with argument plus as in `+/` the result is the derived verb *sum*.
Atom	A 0-dimensional element of an array; it may be numeric, literal or boxed.
Axis	An organizational direction of an array. The shape of an array gives the lengths of the axes of the array.
Cell	A subarray of an array that consists of all the entries from the array with some fixed leading set of indices.
Conjunction	A part of speech that takes two arguments and typically results in a verb. For example, `*:^:3` is a function that iterates squaring three times (`^:` is a conjunction).
Dimension	The dimension of an array is the number of axes given by the array's shape.
Dyad	A verb with two arguments.
Explicit	Describes a definition which uses named arguments; for example, a verb defined using `x` and `y`..
Fork	A list of three verbs isolated in a train so that composition of functions occurs (see Section 18)
Gerund	A verb playing the role of a noun.
Hook	A list of two verbs isolated in a train so that composition of functions occurs (see Section 18)
Inflection	The use of a period or colon suffix to change the meaning of a J word.
Item	A cell of rank `_1`. Thus, an array may be thought of as a list of its items.
Monad	A verb with one argument.
Noun	A data object that is numeric, literal or boxed.
Rank	The dimension of cells upon which a verb operates; additional leading axes are handled uniformly.
Tacit	Function definition without explicit (named) reference to the arguments
Trains	Lists of conjunctions, adverbs, verbs and nouns; for example, a train of three verbs is a fork.
Verb	A function; when it uses two arguments, it is a dyad; and when it uses one argument, it is a monad.

39 Vocabulary

`=` Self-classify • Equal	`=.` Is (local)	`=:` Is (global)
`<` Box • Less Than	`<.` Floor • Lesser Of	`<:` Decrem • Less Or Equal
`>` Open • Larger Than	`>.` Ceiling • Larger Of	`>:` Increm • Larger Or Equal
`_` Negative Sign, *Infinity*	`_.` *Indeterminate*	`_:` Infinity
`+` Conjugate • Plus	`+.` Real/imaginary • GCD (Or)	`+:` Double • Not Or
`*` Signum • Times	`*.` Length/angle • LCM (And)	`*:` Square • Not And
`-` Negate • Minus	`-.` Not (1-) • Less	`-:` Halve • Match
`%` Reciprocal • Divide	`%.` Mat Inv • Mat Divide	`%:` Square Root • Root
`^` Exponential • Power	`^.` Natural Log • Logarithm	`^:` **Power**
`$` Shape Of • Shape	`$.` Sparse	`$:` Self Reference
`~` ***Reflex • Pass, Evoke***	`~.` Nub	`~:` Nub Sieve • Not Equal
`|` Magnitude • Residue	`|.` Reverse • Rotate	`|:` Transpose
`.` **Det • Dot Product**	`..` **Even**	`.:` **Odd**
`:` **Explicit (monad, dyad)**	`:.` **Obverse**	`::` **Adverse**
`,` Ravel • Append	`,.` Ravel Items • Stitch	`,:` Itemize • Laminate
`;` Raze • Link	`;.` **Cut**	`;:` Words • Sequential Machine
`#` Tally • Copy	`#.` Base 2 • Base	`#:` Antibase 2 • Antibase
`!` Factorial • Out Of	`!.` **Fit**	`!:` **Foreign**
`/` ***Insert • Table, Insert***	`/.` ***Oblique • Key, Append***	`/:` Grade Up • Sort Up
`\` ***Prefix • Infix, Train***	`\.` ***Suffix • Outfix***	`\:` Grade Down • Sort Down
`[` Same • Left		`[:` Cap
`]` Same • Right		
`{` Catalogue • From	`{.` Head • Take	`{:` Tail, `{::` Map, Fetch
`}` ***Item Amend • Amend***	`}.` Behead • Drop	`}:` Curtail
`"` **Rank • Constant**	`".` Do • Numbers	`":` Default Format • Format
`` ` `` **Tie (gerund)**		`` `: `` **Evoke Gerund**
`@` **Atop**	`@.` **Agenda**	`@:` **At**
`&` **Bond, Compose**	`&.` `&.:` **Under (Dual)**	`&:` **Appose**
`?` Roll • Deal	`?.` Roll • Deal (fixed seed)	
`a.` *Alphabet*	`a:` *Ace (Boxed Empty)*	`A.` Anagram Index • Anagram
`b.` ***Boolean, Basic***	`C.` Cycle Direct • Permute	`d.` **Derivative**
`D.` **Derivative**	`D:` ***Secant Slope***	`e.` Raze In • Member In
`E.` • Member Of Interval	`f.` ***Fix***	`H.` **Hypergeometric**
`i.` Integers • Index Of	`i:` Axis Integers • Index Of Last	`I.` Indices • Interval Index
`j.` Imaginary • Complex	`L.` Level Verb	`L:` **Level At**
`M.` ***Memo***	`NB.` Comment	`o.` Pi Times • Circle Function
`p.` Roots • Polynomial	`p..` Poly Deriv • Poly Integral	`p:` Primes
`q:` Prime Factors • Prime Exponents	`r.` Angle • Polar	`s:` Symbol
`S:` **Spread**	`t.` ***Taylor Coeff.*** (`m t. u t.`)	`t:` ***Weighted Taylor***
`T.` **Taylor Approximation**	`u:` Unicode	`x:` Extended Precision
`_9: to 9:` Constant Verbs		

Font styles in the Vocabulary: *noun*, verb, ***adverb***, **conjunction**.