A Brief J Reference

by Cliff Reiter
Version 3: July 25, 2002

This brief reference gives informal descriptions of most of the **J** primitives. Not every primitive is included and some phrases, examples and other resources have been added when that seemed appropriate. Since the presentation is so brief and informal, it is not suitable as an introduction to the language and it is not a replacement for the main J references: the *J Introduction and Dictionary*, the *J User manual* and the *J Primer*. Nor is it a replacement for introductions such as Henry Rich's *J for the C programmer*, Roger Stokes' *Learning J*, and Norman Thomson's *J: the Natural Language for Analytic Computing*. However, since the material given here is informally organized by topic, this reference may be useful for brainstorming 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 by turning to the main references.

Table of Contents

1. Basic Arithmetic

`x + y`	`x` *plus* `y`
`+ y`	`y`; that is, for real `y` this is the *identity* function
`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 `y` (for real `y`)
`x % y`	`x` *divided by* `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* of `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`; that is, `y-1` (*decrement of*)
`>: y`	*successor* of `y`; that is, `y+1` (*increment* of)

2. Circular and Numeric Functions

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`	Re(y)	`_9 o. y`	y
`10 o. y`	abs(y) which is `|y`	`_10 o. y`	conjugate(y)
`11 o. y`	Im(y)	`_11 o. y`	*yi* where *i* is $\sqrt{-1}$
`12 o. y`	arg(y)	`_12 o. y`	e^{iy}

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

3. Boolean and Relational Functions

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`; generalizes to `1-y` for numeric `y`.
`x +. y`	`x` *or* `y`; generalizes to the greatest common divisor (*gcd*) of `x` and `y`
`x *. y`	`x` *and* `y`; generalizes to 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; compare to `(; e.&>"_ 0 ])y`
`x E. y`	mark beginnings of list `x` as a sublist in `y` (*pattern occurrence*); see cut in

Section 7 and the regex laboratories for matching more complex patterns than those handled by `E.`.

The 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 bit.

4. Assignment of Names

`abc=: 1 2 3`	*global assignment* of `1 2 3` to the name "`abc`" (*is*)
`abc=. 1 2 3`	*local assignment* of `1 2 3` to the name "`abc`"; that is, the value is only available inside the function where it is used
`'abc' =: 1 2 3`	*indirect assignment* of `1 2 3` to the 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`"
`(exp)=: 1 2 3`	the result of the expression `exp` is assigned the values
`names ''`	currently defined names in base locale; loaded by default configuration
`erase 'a b c'`	erases the three objects "`a`", "`b`", and "`c`"
`(4!:5)1`	turns on data collection and yields names changed since last execution of `(4!:5)1`

Several foreign conjunctions of the form `4!:n` deal with names. See the locales lab to learn about using locales to create different locations for global names. Other `4!:n` functions give the type of the name and deal with script names.

5. Array 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 items)
`i. y`	list of *indices* filling an array of shape `y` (*integer*); negative reverses axis
`i: y`	*symmetric arithmetic sequence*; try `i:5` and `i:5j4`
`x F/ y`	*table* of values of `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` (changes a vector into a 1-column matrix)
`,: y`	*itemize*, make `y` into a single item by adding a new length one leading axis
`$. y`	*sparse* matrix representation of `y`

See Section 16 for information about location of items within an array.

6. Array Selection

`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
`(G # ])y`	selects elements of `y` according to Boolean test `G`; thus, `(2&< # ])y` gives the elements of `y` greater than 2.
`I { y`	item at position `I` in `y` (*index* or *from*); arrays `I` give corresponding arrays of items; boxed arrays `I` give all possible combinations of indices along leading axes (empty box gives all possibilities along that axis); boxed boxed arrays randomly access positions.
`x I} y`	`y` *amended* at positions in `I` by data `x`.
`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`	the item in `1 {. y` for non-empty arrays; in general `0{y` (called *head*)
`{: y`	the 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*

7. Data Amalgamation

The important role that data amalgamation facilities play in organizing computations and analyses makes the study of them worthwhile. Because they are powerful, patience, persistence and restudy are recommended.

`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 lists of length `x` in `y` (the lists are *infixes*); negative `x` gives non-overlapping sublists.
`x avg\ y`	gives length `x` *moving averages* of data in `y` (here `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; try `<;.3 i. 4 4`
`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`.
`#/.~ y`	*frequency* of occurrence of items (given by nub) in `y`
`F/. y`	apply `F` to *oblique* lists from `y`. Try `</.i.4 6`

8. 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 steps 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 (function)
`3 : 0`	input mode for general verb; monadic definition followed by an isolated colon, followed by 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 alternatively be a string, a `CRLF` delimited string, a matrix, or a boxed list of strings that give the "program".

`13 : n`	convert to tacit form of a verb if possible

9. Program Flow Control in J

Control structures offer facilities for organizing the order of execution of J expressions. See the "control structures" reference from the Vocabulary (see the on line help) for details. Also consider the following illustrations and comments. First consider the *if* control structure. Note that `elseif.` is also available.

```
signum=: 3 : 0"0         NB. note the use of rank 0
if. y. < 0 do. _1 else.
  if. y.=0 do. 0 else. 1 end.
  end.
)
```

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

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

Consider the *while* control structure. The control word `whilst.` is the same as `while.` except the steps of the loop are executed once before the control condition must hold.

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

```
   sumint 10
55
```

```
   +/@i.@>: 10
55
```

Consider the *for* control structure.

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

```
   sumintb 10
55
```

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 halt function execution.

The `select.` control word allows the execution of an expressions (or expressions) when a prototype object matches those in a given case or cases.

```
atype=: 3 : 0
select. 3<.#$y.
case. 0 do. 'scalar'
case. 1 do. 'vector'
case. 2 do. 'matrix'
case. 3 do. 'array of dimension greater than 2'
end.
)
```

```
   atype 'abc'
vector
```

```
   atype i.3 3
matrix
```

```
   atype <i.3 3
scalar
```

```
   atype i.3 3 3 3 3
array of dimension greater than 2
```

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.`.

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

10. Reading and Writing Files

These are all based on the foreign conjunctions of the form `1!:n`. These provide for file reading/writing including indexed reads and writes and creating directories, reading and setting attributes and permissions. Convenient utilities are defined in *files.ijs*. Chopping file data in appropriate places can be accomplished with `_2` cut; see Sections 7 and 23. Simple substitution (e.g., "_" for "-") may be accomplished with `charsub` from *strings.ijs*. See *regex.ijs* 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`)
`x 1!:2 y`	write file `y` with character (binary) data `x` (see `fwrite` and `fwrites`)

`x 1!:3 y`	append file `y` with character (binary) data `x` (see `fappend` and `fappends`)

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.

11. Scripts

Scripts consist of text that gives a listing of definitions or **J** expressions to be executed. The text of scripts is often stored in files and these scripts are the natural place to store collections of **J** definitions.

cntl-n	keystroke to open a new script window
`0!:0 <'filename.ijs'`	run the script "*filename.ijs*"; note boxing of filename
`load 'filename.ijs'`	similar to `0!:0` except local definitions made inside the `load` function do not exist upon completion.
`loadd 'filename.ijs'`	similar to `load` except the results of running the script are displayed.
`open 'filename.ijs'`	opens the script in an edit window.
`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

12. Front End Short-Cut Keys

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

enter	grabs 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-tab	shift active window
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)

The following expression would put "My F2" into the tools menu and execute `f2expression` when F2 is pressed.

```
wd 'smsetcmd 2 1 "&My F2',TAB,'F2" "f2expression";'
```

13. 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 appending along an existing axis.
`F&.> y`	apply `F` inside of *each* boxed element of `y`.
`F&> y`	apply `F` to the 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`	the *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 `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`

14. Noun Atoms

`r`	gives rationals; `5r4` is 5/4
`b`	gives base representations; `2b101` is 5
`e`	gives base 10 exponential (scientific notation); `1.2e14` is 1.2×10^{14}
`p`	gives base π exponential notation; `3p6` is $3\pi^6$
`x`	gives base e (natural) exponential notation; `3x2` is $3e^2$
`x`	also gives extended precision; `2^100x` is the exact integer 2^{100}
`j`	gives complex numbers; `3j4` is $3 + 4i$
`ad`	gives *angle in degrees*; `1ad45` is approximately `0.707j0.707`
`ar`	gives *angle in radians*; `1ar1` is `^0j1`
`a.`	*alphabet*: gives the list of all 256 characters including the usual ASCII characters
`a:`	*boxed empty*
`_1`	negative one; negatives denoted with underline prefix
`_`	*infinity*; an underbar in isolation denotes infinity
`__`	*negative infinity*
`_.`	*indeterminant*

15. Conversion: String, Numeric, Base, Binary

`": y`	*format* array `y` as a character array
`ajb ": y`	*format* data in `y` with field width `a` and `b` decimal digits; try `15j10 ": o.i.3 4`
`":!. c y`	*format* data showing `c` decimal figures
`". y`	*execute* or *do* the string `y`

`x ". y`	convert the data in `y` to numeric using `x` for illegal numbers. **J** syntax is relaxed so appearances of `-` in `y` is treated like `_`.
`".@}:;._2 y`	*execute* the expressions in `CRLF` delimited substrings appearing in `y` (assuming it ends with `CRLF`) and adjoin the results
`'m'~`	the value of the name `m` is *evoked*
`#: y`	binary representation of `y` (*antibase-two*)
`x #: y`	representation of `y` base the digit values given in `x` (*antibase*)
`#. y`	value of the binary rank-1 cells of `y` (*base-two*)
`x #. y`	value of the 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. See the foreign conjunction help.

16. Sorting and Searching

`/: y`	*grade up*; indices of items of `y` ordered so that the corresponding elements 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 elements 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; compare `(; e.&>"_ 0 ])y`
`x E. y`	mark beginnings of list `x` as a sublist in `y` (*pattern occurrence*);
`~. y`	*nub* of `y`; that is, items of `y` with duplicates removed
`({.,#)/.~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 elements of `y` according to boolean test `G`; thus, `(2&< # ])y` gives the elements of `y` greater than 2.
`x -. y`	the items of `x` *less* those in `y`

See cut in Section 7 and the regex laboratories for matching more complex patterns than those handled by `E.`.

17. Matrix Arithmetic

`x +/ . * y`	*matrix product* of `x` and `y` (*dot product* for vectors)
`x +/ . = y`	for vectors, gives the number of places where 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`	solution *z* to the linear matrix system *y z* = *x*; least squares solution is given when appropriate. *Matrix divide*.
`%. y`	*matrix inverse* or *pseudo-inverse* of matrix `y`
`|: y`	*transpose* of `y`
`x |: y`	*generalized transpose* of `y`. The 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`	gives a `y` by `y` identity matrix; multiply by diagonal to get a diagonal matrix
`128!:0 y`	*QR decomposition* of `y`

The J AddOns *lapack.ijs* and *fftw.ijs* give extensive linear algebra and fast Fourier transform utilities, respectively.

18. Rank

Rank can be specified by one, two or three elements. If the rank `r` contains three elements, the first is the monadic rank, the second the left dyadic rank and last the right dyadic rank. If it contains two elements, 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 element is given.

`F"r y`	apply `F` on rank `r` cells of the data `y`
`x F"r y`	apply `F` on rank `rr` cells from `x` and rank `lr` cells from `y` where the right rank `rr` and left rank `lr` are specified by `r` as noted above.
`x F"0 _ y`	table builder when `F` is scalar
`N"r`	the constant function of rank `r` and result `N`
`F b. 0`	gives the monadic, left and right ranks of the verb `F`

19. Constant and Identity Functions

`] y`	the result is `y`, the *identity* function on `y` (*same*)
`x ] y`	the result is `y`, the function is called *right*
`[ y`	the result is `y`, the *identity* function on `y` (*same*)
`x [ y`	the result is `x`, the function is called *left*
`0: y`	the result is the scalar 0
`1: y`	the result is 1; likewise, there are constant functions denoted `2:` to `9:` and `_1:` to `_9:`
`_: y`	the result is the infinite scalar _
`N"r`	is the constant function with value `N` on rank `r` cells

20. Function Composition

Atop

```
F@G y                              x F@G y
             F                                   F
             |                                   |
             G                                   G
             |                                  / \
             y                                 x   y
```

Compose

```
F&G y                              x F&G y
             F                                   F
             |                                  / \
             G                                 G   G
             |                                 |   |
             y                                 x   y
```

Under

```
F&.G y                             x F&.G y
             G^-1                                G^-1
             |                                   |
             F                                   F
             |                                  / \
             G                                 G   G
             |                                 |   |
             y                                 x   y
```

Hook

```
(G H) y                            x (G H) y
               G                                    G
              / \                                  / \
             y   H                                x   H
                 |                                    |
                 y                                    y
```

Fork

```
(F G H) y                          x (F G H) y
               G                                    G
              / \                                 /   \
             F   H                               F     H
             |   |                              / \   / \
             y   y                             x   y x   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. Under is `&.:` is the same as `&.` except for rank infinity; that is, `u &.: v` is equivalent to `u&.(v"_)`. 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.

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

21. More Functions From Functions

`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; both known as *bond* or *curry*
`F~ y`	reflects `y` to both arguments; i.e. it is the same as `y F y`; (*reflex*)
`x F~ y`	*pass* interchanges arguments; i.e., it is the same as `y F x`; (*commute*)
`F : G`	function with monad `F` and dyad `G` (*monad/dyad*)
`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`	the *obverse* (*inverse*) of `F`
`F b. 1`	the *identity function* for `F`

22. Gerunds and Controlled Application of Functions

`^:`	*iterate* (*function power*)
`F^:n y`	iterate `F` a total of `n` times on `y`; see the Dictionary for gerund `n`
`F^:_ y`	iterate `F` until convergence (*limit*)
`F^:(i.n)y`	the 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
`H/. y`	evaluate each verb in gerund `H` taken cyclically on data `y` (*evoke gerund*)
``H`:0 y``	alternative form of *evoke gerund* resulting in all combinations of functions from `H` on `y`
`H@.F`	*agenda*: use `F` to select verb from gerund `H` to apply
`F::G`	*adverse*: apply `F`, if an error occurs, apply `G` instead

Many adverbs and conjunctions have gerund meanings that give generalizations; for example, gerund insert cyclically inserts verbs from the gerund. Thus ``+`%/1 2 3 4`` is `1+2%3+4`.

23. Recursion

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

```
   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
```

24. Efficiency, Error Trapping, and Debugging

`6!:2 y`	the *time* (seconds) required to execute the string `y`. Optional left argument specifies the number of repetitions used to obtain average run time
`7!:2 y`	the *space* (bytes) required to execute the string `y`
`u :: v`	gives the result of applying the 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 foreign conjunctions `13!:n` provide debugging facilities. These facilities require a professional license. With the license, running the debug lab is recommended.

25. Randomization and Simulation

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

See also *system\packages\stat\statdist.ijs* for utilities for randomly selecting from various distributions.

26. 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`	the complex number `0jy`; that is, $0 + i\,y$ (*imaginary*)
`x j. y`	the complex number `xjy`; that is, $x + i\,y$ (*complex*)
`+. y`	the pair containing Re(y) and Im(y), (*real/imaginary*)
`*. y`	the polar pair (r,θ) where $\mathrm{y} = re^{i\theta}$, (*length/angle*)
`r. y`	is e^{iy} (*angle to complex*)
`x r. y`	is xe^{iy} (*polar to complex*)

See also the circular functions.

27. Number Theory, Combinatorics and Permutations

`p: y`	the `y`-th *prime* number
`p:^:_1 y`	the number of primes less than `y`
`q: y`	the prime *factors* of `y`
`x q: y`	the prime *factors* of `y` with limited factor base
`x +. y`	the *greatest common divisor* (gcd)
`x *. y`	the *least common multiple* (lcm)
`gcd y`	the 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`	the *residue* of `y` modulo `x` (remainder after division)
`! 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`	gives the *atomic* representation (position) of the permutation `y`
`x A. y`	applies the permutation with atomic representation `x` of order `#y` to `y` (*atomic permute*, *anagram*)
`(i.!n) A. i.n`	gives all permutations of order `n`
`C. y`	gives the *cycle* representation of the numeric permutation `y` as a boxed list; visa versa when `y` is boxed
`x C. y`	*permutes* `y` according to the permutation `x` (either numeric or boxed cyclic representations for the permutation may be used)
`{ y`	*Cartesian* product: all selections of one item from each box in `y`.

28. Exact Integer and Rational Computations

`2x`	is the exact integer 2 (*extended precision*)
`2x^100`	is the exact integer 2^{100}
`2r3`	is the exact rational number $2/3$ (*extended precision*)
`x: y`	convert `y` to extended precision rational
`x:^:_1 y`	convert `y` to fixed precision numeric
`2 x: y`	gives the numerator/denominator of extended precision rationals
`m&|@(2x&^) y`	computes $2^y \bmod m$ efficiently (without computing 2^y)

29. Calculus, Roots and Polynomials

`F D. n y`	the `n`-th derivative of `F` at `y`
`F d. n y`	the `n`-th derivative rank zero: compare to `(F D. 1)"0 y`
`x F D: n y`	the slope of the secant of `F` at `y` and `x+y`
`F t. n`	the `n`-th Taylor series coefficient of `F` about 0
`F t: n`	the `n`-th Taylor series coefficient of `F` about 0 weighted by `!n`
`F T. n y`	the `n`-th degree Taylor polynomial for `F` about 0 evaluated at `y`
`p. y`	polynomial/root; toggles between coefficient representation and the leading-coefficient-with-root boxed representation of polynomials
`x p. y`	polynomial specified by `x` evaluated at points `y`
`p.. y`	Derivative of the polynomial specified by `y`
`x p.. y`	Integral of the polynomial specified by `y` with constant term x

30. Addons

There are several addons available for J; see *http://www.jsoftware.com/download/download.htm*
Also, under windows it is easy to use system dll calls; see the J file
system\packages\winapi\win32api.dat

- *fftw* fast fourier transform package
- *JAR* (or J-ARchive), is a complete database system for J words, data, test cases and general scripts
- *image2* gives facilities for reading and writing 24 bit images in a variety of formats (J4)
- *image3* like *image2*, with significant improvements including some animation and 8-bit support (J5)
- *lapack* standard linear algebra package; nicest documentation I've seen is at *http://www.cs.colorado.edu/~lapack/*
- *SFL* The SFL (Standard Function Library) from iMatix is a portable function library for C/C++ programs (lots of general stuff).

An alternate random number generator by Ralph Selfridge has also appeared on the Jforum but not made it onto the Jsoftware site.

31. 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. While the main features remain, there are significant differences between system graphics for J 4.06 and 5.

Graphics scripts for *fvj2* (available from Cliff's J pages) include:

fvj2\dwin+.ijs	which gives a simple object based window environment
fvj2\raster5.ijs	which contains utilities for working with raster images
fvj2\chaotica+.ijs	which contains utilities for working with chaotic attractors including functions for resolving data in various ways that are useful in a broader contexts.
fvj2\povkit+.ijs	which gives facilities for formatting 3-D scenes for ray tracing by Pov-RAY
fvj2\owin+.ijs	which gives a simple Open GL based 3-D modeling environment.

32. Parts of Speech and Grammar

The words of a string representing a J expression may be obtained using word formation (`;:`). 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 elements and higher dimensional arrays of elements. In general, arrays contain elements that are organized along axes. These arrays may be character, 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-dimentional, 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 "element" 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 their adverbial argument on the left. The derived verb may itself take one argument (where it is a monad) or two 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. (Dot is an isolated period; be careful to distinguish this from a dot immediately after a nonblank symbol that is an inflection.) 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 to obtain the result of an expression is often said to follow a right to left order. 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. Hence, it is probably safer to describe the order of execution by saying that 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 (verbal) expression on its left, namely, `+/` which gives the sum. Thus
`+/ . *` denoted the matrix product.

33. 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, character or boxed.

Axis An organizational direction of an array. The shape of an array gives the lengths 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. Function power `^:` 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 the functions, as described in Section 11 occurs.

Gerund A verb playing the role of a noun.

Hook A list of two verbs isolated so that composition of the functions, as described in Section 11 occurs.

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 (binary) 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

Verb A function; when it uses two arguments, it is a dyad; and when it uses one argument, it is a monad.

Acknowledgements. These notes grew out of Appendix B that appeared in the author's first edition of *Fractals, Visualization and J*. The remarks of Keith Smillie on drafts of these notes were greatly appreciated.

Clifford A. Reiter
Department of Mathematics
Lafayette College
Easton, PA 18042
reiterc@lafayette.edu
http://www.lafayette.edu/~reiterc