

## **Dictionary**

**Roger K. W. Hui**  
**Kenneth E. Iverson**

## J Dictionary

Copyright 1991 - 1998 All Rights Reserved  
Iverson Software Inc.  
33 Major St.  
Toronto, Ontario  
Canada M5S 2K9

[www.jsoftware.com](http://www.jsoftware.com)

ISBN 1-895721-15-6

The J logo is a trademark of Iverson Software Inc.

# Introduction

**J** is a general-purpose programming language available on a wide variety of computers. Although it has a simple structure and is readily learned by anyone familiar with mathematical notions and notation, its distinctive features may make it difficult for anyone familiar with more conventional programming languages.

This introduction is designed to present **J** in a manner that makes it easily accessible to programmers, by emphasizing those aspects that distinguish it from other languages. These include:

1. A mnemonic one- or two-character spelling for primitives.
2. No order-of-execution hierarchy among functions.
3. The systematic use of *ambivalent* functions that, like the minus sign in arithmetic, can denote one function when used with two arguments (*subtraction* in the case of  $-$ ), and another when used with one argument (*negation* in the case of  $-$ ).
4. The adoption of terms from English grammar that better fit the grammar of **J** than do the terms commonly used in mathematics and in programming languages. Thus, a function such as addition is also called a *verb* (because it performs an action), and an entity that modifies a verb (not available in most programming languages) is accordingly called an *adverb*.
5. The systematic use of adverbs and conjunctions to modify verbs, so as to provide a rich set of operations based upon a rather small set of verbs. For example,  $+/a$  denotes the sum over a list  $a$ , and  $*/a$  denotes the product over  $a$ , and  $a */ b$  is the multiplication table of  $a$  and  $b$ .
6. The treatment of vectors, matrices, and other arrays as single entities.
7. The use of *functional* or *tacit* programming that requires no explicit mention of the arguments of a function (program) being defined, and the use of assignment to assign names to functions (as in `sum=:+/` and `mean=:sum % #`).

The following sections are records of actual **J** sessions, accompanied by commentary that should be read only after studying the corresponding session (and perhaps experimenting with variations on the computer). The sections should be studied with the **J** system and dictionary at hand, and the exercises should be attempted. The reckless reader may go directly to the sample topics.

## 1. Mnemonics

The left side of the page shows an actual computer session with the result of each sentence shown at the left margin. First cover the comments at the right, and then attempt to state in English the meaning of each primitive so as to make clear the relations between related symbols. For example, “< is *less than*” and “<. is *lesser of* (that is, minimum)”. Then uncover the comments and compare with your own.

0	7<5	Less than A zero is interpreted as <i>false</i> .
5	7<.5	Lesser of
1	7>5	Greater than A one is <i>true</i> ( <i>à la</i> George Boole)
7	7>.5	Greater of
1000	10^3	Power ( <i>à la</i> Augustus De Morgan)
3	10^.1000	Logarithm
0	7=5	Equals
5	b=: 5 7<.b	Is ( <i>assignment</i> or <i>copula</i> )
	Min=: <. power=: ^ gt=: >	Min is <. power is ^ gt is >
1000	10 power (5 Min 3)	

### Exercises

- 1.1 Enter the following sentences on the computer, observe the results, give suitable names to any new primitives (such as `*` and `+`, and `*.`), and comment on their behaviour.

```
a=:0 1 2 3
b=:3 2 1 0
a+b
a*b

a-b
a%b
a^b
a^.b

a<b
a>b
(a<b)+(a>b)
(a<b)+.(a>b)
```

Compare your comments with the following:

- a) Negative 3 is denoted by `_3`. The underbar `_` is part of the representation of a negative number in the same sense that the decimal point is part of the representation of one-half when written in the form `0.5`, and the negative sign `_` must not be confused with the symbol used to denote subtraction (i.e., `-`).
  - b) Division (`%`) by zero yields infinity, denoted by the underbar alone.
  - c) Log of 2 to the base 1 is infinite, and log of 0 to the base 3 is negative infinity (`__`).
  - d) Since the relation `5<7` is true, and the result of `5<7` is `1`, it may be said that true and false are represented by the ordinary integers `1` and `0`. George Boole used this same convention, together with the symbol `+` to represent the *boolean* function *or*. We use the distinct representation `+` to avoid conflict with the analogous (but different) *addition* (denoted by `+`).
- 1.2 Following the example `Min=:<.`, invent, assign, and use names for each of the primitives encountered thus far.

## 2. Ambivalence

Cover the comments on the right and provide your own.

```

7-5
2
-5
_5

```

The function in the sentence `7-5` applies to two arguments to perform subtraction, but in the sentence `-5` it applies to a single argument to perform negation.

Adopting from chemistry the term valence, we say that the symbol `-` is *ambivalent*, its effective binding power being determined by context.

```

7%5
1.4
%5
0.2

```

The ambivalence of `-` is familiar in arithmetic; it is here extended to other functions.

```

3^2
9
^2
7.38906

```

*Exponential* (that is,  $2.71828^2$ )

```

a=: i. 5
a
0 1 2 3 4
a i. 3 1
3 1

```

The function *integer* or *integer list*

*List* or *vector*

The function *index* or *index of*

```

b=: 'Canada'
b i. 'da'
4 1
$a
5

```

Enclosing quotes denote literal characters

*Shape* function

```

3 4 $ a
0 1 2 3
4 0 1 2
3 4 0 1

```

*Reshape* function

*Table* or *matrix*

```

3 4 $ b
Cana
daCa
nada

```

```

%a
_ 1 0.5 0.333333 0.25

```

Functions apply to lists

The symbol `_` alone denotes *infinity*

## Exercises

- 2.1 Enter the following sentences (and perhaps related sentences using different arguments), observe the results, and state what the two cases (*monadic* and *dyadic*) of each function do:

```
a=: 3 1 4 1 5 9
b=: 'Canada'
#a
1 0 1 0 1 3 # a
1 0 1 0 1 3 # b
/: a
/: b
a /: a
a /: b
b /: a
b /: b
c=: 'can' 't'
c
#c
c /: c
```

- 2.2 Make a summary table of the functions used thus far. Then compare it with the following table (in which a bullet separates the monadic case from the dyadic, as in Negate • Subtract):

	•	:
+	• Add	• Or
-	Negate • Subtract	
*	• Times	• And
%	Reciprocal • Divide	
^	Exponential • Power	• Log
<	• Less Than	• Lesser Of
>	• Greater Than	• Greater Of
=	• Equals	Is (Copula)
i		Integers • Index Of
\$	Shape • Reshape	
/		Grade • Sort
#	Number of items • Replicate	

- 2.3 Try to fill some of the gaps in the table of Exercise 2.2 by experimenting on the computer with appropriate expressions. For example, enter `^.10` and `^. 2.71828` to determine the missing (monadic) case of `^.`  and enter `%: 4` and `%: -4` and `+%: -4` to determine the case of `%` followed by a colon.

However, do not waste time on matters (such as, perhaps, complex numbers or the *boxed* results produced by the monad `<`) that are still beyond your grasp; it may be better to return to them after working through later sections. Note that the effects of certain functions become evident only when applied to arguments other than positive integers: try `<.1 2 3 4` and `<.3.4 5.2 3.6` to determine the effect of the monad `<.` .

- 2.4 If `b=: 3.4 5.2 3.6`, then `<.b` yields the argument `b` rounded *down* to the nearest integer. Write and test a sentence that rounds the argument `b` to the *nearest* integer.

Answer: `<.(b+0.5)` or `<.b+0.5` or `<.b+1r2`

- 2.5 Enter `2 4 3 $ i. 5` to see an example of a *rank 3 array* or *report* (for two years of four quarters of three months each).

- 2.6 Enter `?9` repeatedly and state what the function `?` does. Then enter `t=: ?3 5 $ 9` to make a table for use in further experiments.

Answer: `?` is a (pseudo-) random number generator;  
`?n` produces an element from the population `i.n`



### 3. Verbs and Adverbs

In the sentence %a of Section 2, the % “acts upon” a to produce a result, and %a is therefore analogous to the notion in English of a *verb* acting upon a noun or pronoun. We will hereafter adopt the term *verb* instead of (or in addition to) the mathematical term *function* used thus far.

The sentence +/ 1 2 3 4 is equivalent to  $1+2+3+4$ ; the adverb / applies to *its* verb argument + to produce a new verb whose argument is 1 2 3 4, and which is defined by inserting the verb + between the items of its argument. Other arguments of the insert adverb are treated similarly:

```
* /b=:2 7 1 8 2 8
1792
```

```
< . /b
1
```

```
> . /b
8
```

The verb resulting from the application of an adverb may (like a primitive verb) have both monadic and dyadic cases; due to its two uses, the adverb / is called either *insert* or *table*. In the present instance of / the dyadic case produces a *table*. For example:

```
2 3 5 +/ 0 1 2 3
2 3 4 5
3 4 5 6
5 6 7 8
```

The verbs over=:({.;}.)@":@, and by=: ' '&;@, .@[ , . ] can be entered as utilities (for use rather than for immediate study), and can clarify the interpretation of *function tables* such as the addition table produced above. For example:

```
a=: 2 3 5
b=: 0 1 2 3
a by b over a +/ b
```

	0	1	2	3
2	2	3	4	5
3	3	4	5	6
5	5	6	7	8

```
b by b over b </ b
```

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

### Exercises

- 3.1 Enter  $d =: i.5$  and the sentences  $st =: d - / d$  and  $pt =: d ^ / d$  to produce function tables for subtraction and power.
- 3.2 Make tables for further functions from previous sections, including the relations  $<$  and  $=$  and  $>$  and *lesser-of* and *greater-of*.
- 3.3 Apply the verbs  $|.$  and  $|:$  to various tables, and try to state what they do.
- 3.4 The *transpose* function  $|:$  changes the subtraction table, but appears to have no effect on the multiplication table. State the property of those functions whose tables remain unchanged when transposed.

Answer: They are commutative

- 3.5 Enter  $d$  by  $d$  over  $d! / d$  and state the definition of the dyad  $!$ .

Answer:  $!$  is the binomial coefficient or *outof* function;  $3!5$  is the number of ways that three things can be chosen from five.

#### 4. Punctuation

English employs various symbols to *punctuate* a sentence, to indicate the order in which its phrases are to be interpreted. Thus:

The teacher said he was stupid.  
The teacher, said he, was stupid.

Math also employs various devices (primarily parentheses) to specify order of interpretation or, as it is usually called, *order of execution*. It also employs a set of rules for an unparenthesized phrase, including a hierarchy amongst functions. For example, *power* is executed before *times*, which is executed before *addition*.

**J** uses only parentheses for punctuation, together with the following rules for unparenthesized phrases:

The right argument of a verb is the value of the entire phrase to its right.  
Adverbs are applied first. Thus, the phrase  $a * / b$  is equivalent to  $a (* / ) b$ , not to  $a (* / b)$ .

For example:

$a = : 5$   
 $b = : 3$   
 $(a * a) + (b * b)$   
34  
 $a * a + b * b$   
70  
 $a * (a + (b * b))$   
70  
 $(a + b) * (a - b)$   
16  
 $a (+ * -) b$   
16

The last sentence above includes the *isolated* phrase  $+ * -$  which has thus far not been assigned a meaning. It is called a *trident* or *fork*, and is equivalent to the sentence that precedes it.

12

A fork also has a monadic meaning, as illustrated for the *mean* below:

$c = : 2 \ 3 \ 4 \ 5 \ 6$   
4  $(+/ \% \#) \ c$       The verb  $\#$  yields the number of items in its argument  
  
4  $(+/c) \% (\#c)$

### Exercises

4.1 In math, the expression  $3x^4 + 4x^3 + 5x^2$  is called a *polynomial*. Enter:

$x = : 2$   
 $(3 * x^4) + (4 * x^3) + (5 * x^2)$

to evaluate the polynomial for the case where  $x$  is 2.

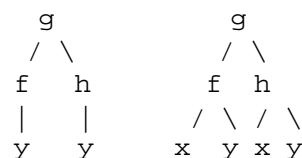
4.2 Note that the hierarchy among functions used in math is such that no parentheses are necessary in writing a polynomial. Write an equivalent sentence using no parentheses.

Answer:  $+ / 3 \ 4 \ 5 \ * \ x \ ^ \ 4 \ 3 \ 2$

or (first assigning names to the *coefficients* 3 4 5 and the *exponents* 4 3 2), as:  $+ / c * x^e$

## 5. Forks

As illustrated above, an isolated sequence of three verbs is called a *fork*; its monadic and dyadic cases are defined by:



$$\begin{aligned}
 (f \ g \ h) \ y &\leftrightarrow (f \ y) \ g \ (h \ y) \\
 x \ (f \ g \ h) \ y &\leftrightarrow (x \ f \ y) \ g \ (x \ h \ y)
 \end{aligned}$$

The diagrams at the upper right provide visual images of the fork.

Before reading the notes at the right (and by using the facts that  $\% :$  denotes the *root* function and  $[]$  denotes the *identity* function), try to state in English the significance of each of the following sentences:

$$\begin{aligned}
 a &=: 8 \ 7 \ 6 \ 5 \ 4 \ 3 \\
 b &=: 4 \ 5 \ 6 \ 7 \ 8 \ 9 \\
 2 \ \% : b &\qquad\qquad\qquad \text{Square root of } b \\
 2 \ 2.23607 \ 2.44949 \ 2.64575 \ 2.82843 \ 3
 \end{aligned}$$

$$\begin{aligned}
 3 \ \% : b &\qquad\qquad\qquad \text{Cube root of } b \\
 1.5874 \ 1.70998 \ 1.81712 \ 1.91293 \ 2 \ 2.08008
 \end{aligned}$$

$$\begin{aligned}
 (+/\ \% \#) \ b &\qquad\qquad\qquad \text{Arithmetic mean or average} \\
 6.5
 \end{aligned}$$

$$\begin{aligned}
 (\# \% : */) \ b &\qquad\qquad\qquad \text{Geometric mean} \\
 6.26521
 \end{aligned}$$

$$\begin{aligned}
 ([ - (+/\ \% \#)) \ b &\qquad\qquad\qquad \text{Centre on mean (two forks)} \\
 \_2.5 \ \_1.5 \ \_0.5 \ 0.5 \ 1.5 \ 2.5
 \end{aligned}$$

$$\begin{aligned}
 ([ - +/\ \% \#) \ b &\qquad\qquad\qquad \text{Two forks (fewer parentheses)} \\
 \_2.5 \ \_1.5 \ \_0.5 \ 0.5 \ 1.5 \ 2.5
 \end{aligned}$$

$$\begin{aligned}
 a \ (+ \ * \ -) \ b &\qquad\qquad\qquad \text{Dyadic case of fork} \\
 48 \ 24 \ 0 \ \_24 \ \_48 \ \_72
 \end{aligned}$$

$$\begin{aligned}
 (a^2) - (b^2) &\qquad\qquad\qquad \\
 48 \ 24 \ 0 \ \_24 \ \_48 \ \_72
 \end{aligned}$$

$$\begin{aligned}
 a \ (< \ +. \ =) \ b &\qquad\qquad\qquad \text{Less than or equal}
 \end{aligned}$$

14

0 0 1 1 1 1

```

a<b
0 0 0 1 1 1
a=b
0 0 1 0 0 0

```

```

a (<: = < +. =) b
1 1 1 1 1 1

```

A tautology (<: is *less than or equal*)

```

2 ([: ^ -) 0 1 2
7.38906 2.71828 1

```

Cap yields monadic case

```

evens=: [: +: i.
evens 7
0 2 4 6 8 10 12

```

+: is *double*

```

odds=: [: >: evens
odds 7
1 3 5 7 9 11 13

```

>: is *increment*

### Exercises

- 5.1 Enter `5#3` and similar expressions to determine the definition of the dyad `#` and then state the meaning of the following sentence:

```
(# # >./) b=: 2 7 1 8 2
```

Answer: `#b` repetitions of the maximum over `b`

- 5.2 Cover the comments on the right, write your own interpretation of each sentence, and then compare your statements with those on the right:

```

(+ / % #) b
(# # + / % #) b
+ / (## + / % #) b
(+ / b) = + / (## + / % #) b
(* / b) = * / (### % : * /) b

```

Mean of `b`  
 (`n=: #b`) repetitions of mean  
 Sum of `n` means  
 Tautology  
 The product over `b` is the product over `n` repetitions of the geometric mean of `b`.

## 6. Programs

A *program* handed out at a musical evening describes the sequence of musical pieces to be performed. As suggested by its roots *gram* and *pro*, a program is something written in advance of the events it prescribes.

Similarly, the fork `+ / % #` of the preceding section is a program that prescribes the computation of the mean of its argument when it is applied, as in the sentence `(+ / % #) b`. However, we would not normally call the procedure a program until we assign a name to it, as illustrated below:

```
mean=: + / % #
mean 2 3 4 5 6
4

(geomean=: # %: */ ) 2 3 4 5 6
3.72792
```

Since the program `mean` is a new *verb*, we also refer to a sentence such as `mean=: + / % #` as *verb definition* (or *definition*), and to the resulting verb as a *defined* verb or function.

Defined verbs can be used in the definition of further verbs in a manner sometimes referred to as *structured programming*. For example:

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

MEAN 2 3 4 5 6
4
```

Entry of a verb alone (without an argument) displays its definition, and the foreign conjunction `(! :)` can be used to specify the form of the display: boxed, tree, linear, or parens. (The session can also be configured to specify the form of verb display, under the menu item Edit|Configure...|View.) Thus:



```

mean
+ / % #
9 ! : 3 ( 2 4 5 )
mean
+-----+-----+
| +--+--+ | % | # | | |
| | + | / | | |
| +--+--+ | | |
+-----+-----+
+- / --- +
--- %
+- #
+ / % #

```

### Exercises

- 6.1 Enter `AT=: i. +/ i.` and use expressions such as `AT 5` to determine the behaviour of the program `AT`.
- 6.2 Define and use similar function tables for other dyadic functions.
- 6.3 Define the programs:  
`tab=: +/`  
`ft=: i. tab i.`  
`test1=: ft = AT`
- Then apply `test1` to various integer arguments to test the proposition that `ft` is equivalent to `AT` of Exercise 7.1, and enter `ft` and `AT` alone to display their definitions.
- 6.4 Define `aft=: ft f.` and use `test2=: aft = ft` to test their equivalence. Then display their definitions and state the effect of the adverb `f.`
- Answer: The adverb `f.` *fixes* the verb to which it applies, replacing each name used by its definition.
- 6.5 Redefine `tab` of Exercise 7.3 by entering `tab=: */` and observe the effects on `ft` and its *fixed* alternative `aft`.
- 6.6 Define `mean=: +/ % #` and state its behaviour when applied to a *table*, as in `mean t=: (i. !/ i.) 5`.
- Answer: The result is the average *of*, not *over*, the rows of a table argument.
- 6.7 Write an expression for the mean of the *columns* of `t`.
- Answer: `mean |: t` or `mean"1 t`

## 7. Bond Conjunction

A dyad such as `^` can be used to provide a family of monadic functions:

```

]b=: i.7
0 1 2 3 4 5 6

    b^2
0 1 4 9 16 25 36
Squares

    b^3
0 1 8 27 64 125 216
Cubes

    b^0.5
0 1 1.41421 1.73205 2 2.23607 2.44949
Square roots

```

The *bond* conjunction `&` can be used to bind an argument to a dyad in order to produce a corresponding defined verb. For example:

```

square=: ^&2
square b
0 1 4 9 16 25 36
Square (power and 2)

(sqrt=: ^&0.5) b
0 1 1.41421 1.73205 2 2.23607 2.44949
Square root function

```

A left argument can be similarly bound:

```

Log=: 10&^.
Log 2 4 6 8 10 100 1000
0.30103 0.60206 0.778151 0.90309 1 2 3
Base-10 logarithm

```

Such defined verbs can of course be used in forks. For example:

```

in29=: 2<& * . <&9
in29 0 1 2 5 8 13 21
0 0 0 1 1 0 0
Interval test

IN29=: in29 # ]
IN29 0 1 2 5 8 13 21
5 8
Interval selection

LOE=: <+. =
5 LOE 3 4 5 6 7

```

0 0 1 1 1

```

integertest=: <. = ]           The monad <. is floor
integertest 0 0.5 1 1.5 2 2.5 3
1 0 1 0 1 0 1

int=: integertest
int (i.13)%3
1 0 0 1 0 0 1 0 0 1 0 0 1

```

### Exercises

- 7.1 The verb `#` is used dyadically in the program `IN29`. Enter expressions such as `(j=: 3 0 4 0 1) # i.5` to determine the behaviour of `#`, and state the result of `#j#i.5`. (Also try `1j1#i.5`.)

Answer: `+/j`

- 7.2 Cover the answers on the right and apply the following programs to lists to determine (and state in English) the purpose of each:

<code>test1=: &gt;&amp;10 *. &lt;&amp;100</code>	Test if in 10 to 100
<code>int=: ] = &lt;</code>	Test if integer
<code>test2=: int *. test1</code>	Test if integer and in 10 to 100
<code>test3=: int +. test1</code>	Test if integer or in 10 to 100
<code>sel=: test2 # ]</code>	Select integers in 10 to 100

- 7.3 Cover the program definitions on the left of the preceding exercise, and make new programs for the stated effects.
- 7.4 Review the use of the *fix* adverb in Exercises 5.4-5, and experiment with its use on the programs of Exercise 6.2.

## 8. Atop Conjunction

The conjunction @ applies to two verbs to produce a verb that is equivalent to applying the first atop the second. For example:

```
TriplePowersOf2=: (3&*)@(2&^)
TriplePowersOf2 0 1 2 3 4
3 6 12 24 48
```

```
CubeOfDiff=: (^&3)@-
3 4 5 6 CubeOfDiff 6 5 4 3
_27 _1 1 27
```

```
f=: ^@-
5 f 3
7.38906
```

The rightmost function is first applied dyadically if possible; the second is applied monadically.

```
f 3
0.0497871
```

```
g=: -@^
5 g 3
_125
```

```
g 3
_20.0855
```

A conjunction, like an adverb, is executed before verbs; the *left* argument of either is the entire verb phrase that precedes it. Consequently, some (but not all) of the parentheses in the foregoing definitions can be omitted. For example:

```
COD=: ^&3@-
3 4 5 6 COD 6 5 4 3
_27 _1 1 27
```

```
TPO2=: 3&*(2&^)
TPO2 0 1 2 3 4
3 6 12 24 48
```

```
tpo2=: 3&*@2&^
|domain error
| tpo2=: 3&*@2&^
```

An error because the conjunction @ is defined only for a verb right argument

**Exercises**

- 8.1 Cover the comments on the right, and state the effects of the programs. Then cover the programs and rewrite them from the English statements:

<code>mc=: (+/%#)@  :</code>	Means of columns of table
<code>f=: +/ @ (^&amp;2)</code>	Sum of squares of list
<code>g=: %:@f</code>	Geometric length of list
<code>h=: {&amp;' *'@(&lt;/)</code>	Map of comparison (dyad)
<code>k=: i. h i.</code>	Map (monad)
<code>map=: {&amp;' +-* %#\$'</code>	7-character map
<code>MAP=: map@ (6&amp;&lt;.)</code>	Extended domain of map
<code>add=: MAP@ (i.+/i.)</code>	Addition table map

## 9. Vocabulary

Memorizing lists of words is a tedious and ineffectual way to learn a language, and better techniques should be employed:

- A) Conversation with a native speaker who allows you to do most of the talking.
- B) Reading material of interest in its own right.
- C) Learning how to use dictionaries and grammars so as to become independent of teachers.
- D) Attempting to *write* on any topic of interest in itself.
- E) Paying attention to the *structure* of words so that known words will provide clues to the unknown. For example, *program* (already analyzed) is related to *tele* (far off) *gram*, which is in turn related to *telephone*. Even tiny words may possess informative structure: *atom* means not cuttable, from *a* (not) and *tom* (as in *tome* and *microtome*).

In the case of **J**:

- A) The computer provides for precise and general conversation.
- B) Texts such as *Fractals, Visualization and J* [7], *Exploring Math* [8], and *Concrete Math Companion* [14] use the language in a variety of topics.
- C) The appended dictionary of **J** provides a complete and concise dictionary and grammar.
- D) *J Phrases* [9] provides guidance in writing programs, and almost any topic provides problems of a wide range of difficulty.
- E) Words possess considerable structure, as in `+:` and `-:` and `*:` and `%:` for *double*, *halve*, *square*, and *square root*. Moreover, a beginner can assign and use mnemonic names appropriate to any native language, as in `sqrt=: %:` and `entier=: <.` (French name) and `sin=: 1&o.` and `SIN=: 1&o. @ (%&180@o.)` (for sine in degrees).

We will hereafter introduce and use new primitives with little or no discussion, assuming that the reader will experiment with them on the computer, consult the dictionary to determine their meanings, or perhaps infer their meanings from their structure. For example, the appearance of the word `o.` suggests a circle; it was used dyadically above to define the sine (one of the *circular* functions), and monadically for the function *pi times*, that is, the circumference of a *circle* when applied to its diameter.





For precise oral communication it may be best to use the names (or abbreviations) of the symbols themselves, as in:

<	Left a (angle)	/	Slash	&	Amp (ersand)	%	Per( cent)
[	Left b (racket)	\	Back (slash)	@	At	;	Semi (colon)
{	Left c (urly bracket)		Stile	^	Caret	~	Tilde
(	Left p (arenthesis)	_	(Under) Bar	`	Grave	*	Star

### Exercises

- 9.1 Experiment with a revised version of the program `MAP` of Exercise 7.1, using the *remainder* or *residue* dyad (`|`) instead of the *minimum* (`< .`), as in `M=:map@ ( 6&| )` and compare its results with those of `MAP`.
- 9.2 Experiment with the programs `sin` and `SIN` defined in this section.
- 9.3 Write programs using various new primitives found in the vocabulary at the end of the book.
- 9.4 Update the table of notation prepared in Exercise 2.2.

## 10. Housekeeping

In an extended session it may be difficult to remember the names assigned to verbs and nouns; the *foreign* conjunction `!` (detailed in the appendix) provides facilities for displaying and erasing them. For example:

```

b=: 3* a=: i. 6
sum=: +/
tri=: sum\ a
names=: 4!:1

names 0
+--+-----+
|a|b|tri|
+--+-----+

names 3
+-----+-----+
|names|sum|
+-----+-----+

erase=: 4!:55
erase <'tri'
1

names 0 3
+--+-----+-----+-----+
|a|b|erase|names|sum|
+--+-----+-----+-----+

erase names 0
1 1

names 0 3
+-----+-----+-----+
|erase|names|sum|
+-----+-----+-----+

```

The Windows drop-down menus can be used to save, retrieve, and print sessions. They can also be used to open *script windows*, in which any number of sentences may be entered and edited, and from which they can be *executed* so as to appear in the normal execution window.

The script and execution windows can also be displayed side-by-side, so that the effect of executing scripts can be directly observed.

These matters are treated in detail in the help files, as are the housekeeping facilities provided by the script file *profile.ijs*. The menus may be used to execute this file explicitly so as to make its facilities available. If, however, its name is included in the *command line* (as described in the user manual) it is executed automatically at the beginning of a session.

### Exercises

- 10.1 Enter and experiment with the programs defined in this section.
- 10.2 Type in the sentence `+ / 2 3 5 * i . 3` and press the Enter key to execute it.

The following exercises describe facilities available only under Windows and Macintosh.

- 10.3 Use the Up-arrow cursor key to move the cursor back up the line entered in Exercise 10.2, and then:
- Press Enter to bring the line down to the input area.
- Use the Left-arrow key to move the cursor back to the `*` symbol, and the backspace key to erase it.
- Enter `-` to replace the multiplication by subtraction, and press the Enter key to execute the revised sentence.
- 10.4 Use cursor keys to move the cursor to the immediate left of the `i` in the sentence executed in Exercise 10.3. Then hold down the Control key and press the F1 key to display Dictionary definition of the primitive `i`.
- 10.5 Press the Escape key to close the display invoked in Exercise 10.4. Then move the cursor to the left so that it is separated from the line by one or more spaces, and again perform Ctrl F1 to display the individually boxed words in the sentence.

## 11. Power and Inverse

The power conjunction ( $\wedge$ ) is analogous to the power function ( $\wedge$ ). For example:

```

]a=: 10^ b=: i.5
1 10 100 1000 10000

      b
0 1 2 3 4

%:a
1 3.16228 10 31.6228 100

%: %: a
1 1.77828 3.16228 5.62341 10

%: ^: 2 a
1 1.77828 3.16228 5.62341 10

%: ^: 3 a
1 1.33352 1.77828 2.37137 3.16228

%: ^: b a
1      10      100      1000      10000
1 3.16228      10 31.6228      100
1 1.77828 3.16228 5.62341      10
1 1.33352 1.77828 2.37137 3.16228
1 1.15478 1.33352 1.53993 1.77828

(cos=: 2&o.) ^: b d=:1
1 0.540302 0.857553 0.65429 0.79348

] y=: cos ^: _ d
0.739085

y=:cos y
1

```

Successive applications of `cos` appear to be converging to a limiting value; the infinite power (`cos ^: _`) yields this limit.

A right argument of `_1` produces the *inverse* function. Thus:

```
%: ^: _1 b
0 1 4 9 16
```

```
*: b
0 1 4 9 16
```

```
%: ^: (-b) b
0 1      2      3      4
0 1      4      9      16
0 1      16     81     256
0 1     256    6561    65536
0 1 65536 4.30467e7 4.29497e9
```

### Exercises

- 11.1 The square function `*:` is the inverse of the square root function `%:` and `%: ^: _1` is therefore equivalent to `*:`. Look for, and experiment with, other inverse pairs among the primitive functions in the Vocabulary.

## 12. Reading and Writing

Translating to and from a known language provides useful beginning exercises in learning a new one. The following provides examples.

**Cover** the **right** side of the page and make a serious attempt to translate the sentences on the **left** to English; that is, state succinctly in English what the verb defined by each sentence does. Use any available aids, including the dictionary and experimentation on the computer:

f1=: <:	Decrement (monad); Less or equal
f2=: f1&9	Less or equal 9
f3=: f2 *. >:&2	Interval test 2 to 9 (inclusive)
f4=: f3 *. <. = ]	In 2 to 9 and integer
f5=: f3 +. <. = ]	In 2 to 9 or integer
g1=: %&1.8	Divide by 1.8
g2=: g1^:_1	Multiply by 1.8
g3=: -&32	Subtract 32
g4=: g3^:_1	Add 32
g5=: g1@g3	Celsius from Fahrenheit
g6=: g5^:_1	Fahrenheit from Celsius
h1=: >./	Maximum over list (monad)
h2=: h1-<./	Spread. Try h2 b with the parabola: b=: (-&2 * -&3) -:i.12
h3=: h1@]-i.@[*h2@]%<:@[	Grid. Try 10 h3 b
h4=: h3 <:/ ]	Barchart. Try 10 h4 b
h5=: {&' *' @ h4	Barchart. Try 10 h5 b

After entering the foregoing definitions, enter each verb name alone to display its definition, and learn to interpret the resulting displays. To see several forms of display, first enter 9!:3 (2 4 5).

**Cover** the **left** side of the page, and translate the English definitions on the **right** back into **J.**

## Exercises

- 12.1 These exercises are grouped by topic and organized like the section, with programs that are first to be read and then to be rewritten. However, a reader already familiar with a given topic might begin by writing.

### A. Properties of numbers

<code>pn=: &gt;:@i.</code>	Positive numbers (e.g. <code>pn 9</code> )
<code>rt=: pn   / pn</code>	Remainder table
<code>dt=: 0&amp;=@rt</code>	Divisibility table
<code>nd=: +/@dt</code>	Number of divisors
<code>prt=: 2&amp;=@nd</code>	Prime test
<code>prsel=: prt # pn</code>	Prime select
<code>N=: &gt;:@pn</code>	Numbers greater than 1
<code>rtt=: ,@(N */ N)</code>	Ravelled times table
<code>aprt=: -.@(N e. rtt)</code>	Alternative test and selection (primes do not occur in the times table for <code>N</code> )
<code>apsel=: aprt # N</code>	
<code>pfac=: q:</code>	Prime factors
<code>first=: p:@i.</code>	First primes

### B. Coordinate Geometry

Do experiments on the vector (or *point*) `p=: 3 4` and the triangle represented by the table `tri=: 3 2$ 3 4 6 5 7 2`

<code>L=: %:@:(+ / )@: * : "1</code>	Length of vector (See <i>rank</i> in the dictionary or in Section 20 of the Introduction)
<code>LR=: L "1</code>	Length of rows in table
<code>disp=: ] - 1&amp; . </code>	Displacement between rows
<code>LS=: LR@disp</code>	Lengths of sides of figure
<code>sp=: -:@(+ / )@LS</code>	Semiperimeter (try <code>sp tri</code> )
<code>H=: %:@( * / )@ ( sp , sp - LS )</code>	Heron's formula for area
<code>det=: - / . *</code>	Determinant (See dictionary)
<code>SA=: det@ ( , .&amp;0.5 )</code>	Signed area. Try <code>SA@  .</code>
<code>sa=: det@ ( ] , .%@!@&lt;:@# )</code>	General signed volume; try on the tetrahedron as well as on the triangle <code>tri</code> .
<code>tet=: ?4 3\$9</code>	

### 13. Format

A numeric table such as:

```

]t=: (i.4 5)%3
      0 0.333333 0.666667      1 1.33333
1.66667      2 2.33333 2.66667      3
3.33333 3.66667      4 4.33333 4.66667
      5 5.33333 5.66667      6 6.33333

```

can be rendered more readable by *formatting* it to appear with a specified width for each column, and with a specified number of digits following the decimal point. For example:

```

]f=: 6j2 " : t
0.00 0.33 0.67 1.00 1.33
1.67 2.00 2.33 2.67 3.00
3.33 3.67 4.00 4.33 4.67
5.00 5.33 5.67 6.00 6.33

```

The real part of the left argument of the format function specifies the column width, and the imaginary part specifies the number of digits to follow the decimal point.

Although the formatted table *looks* much like the original table `t`, it is a table of *characters*, not of numbers. For example:

```

$ t
4 5

$ f
4 30

+/t
10 11.3333 12.6667 14 15.3333

+/f
|domain error
|      +/f

```

However, the verb *do* or *execute* (`" .`) applied to such a character table yields a corresponding numeric table:

```

" . f
0 0.33 0.67      1 1.33
1.67      2 2.33 2.67      3
3.33 3.67      4 4.33 4.67

```



```

5 5.33 5.67      6 6.33
+/" f
10 11.33 12.67 14 15.33

```

### Exercises

13.1 Using the programs defined in Section 12, experiment with the following expressions:

```

5j2 ": d=: %: i.12
5j2 ":",.d
fc=: 5j2&" :@,.
fc d
20 (fc@h3 ,. h5) d
20 (fc@h3 ,. '|'&,.@h5) d
plot=: fc@h3,.'|'&,.@h5
20 plot d

```

## 14. Partitions

The function `sum=: +/` applies to an entire list argument; to compute *partial sums* or *subtotals*, it is necessary to apply it to each prefix of the argument. For example:

```
sum=: +/ [ . a=: 1 2 3 4 5 6
(sum a) ; (sum\ a)
+---+-----+
|21|1 3 6 10 15 21|
+---+-----+
```

The symbol `\` denotes the *prefix* adverb, which applies its argument (in this case `sum`) to each prefix of the eventual argument. The adverb `\.` applies similarly to suffixes:

```
sum\ . a
21 20 18 15 11 6
```

The monad `<` simply *boxes* its arguments, and the verbs `<\` and `<\.` therefore show the effects of partitions with great clarity. For example:

```
<1 2 3
+-----+
|1 2 3|
+-----+

(<1) , (<1 2) , (<1 2 3)
+---+---+---+
|1|1 2|1 2 3|
+---+---+---+

<\ a
+---+---+---+---+---+---+
|1|1 2|1 2 3|1 2 3 4|1 2 3 4 5|1 2 3 4 5 6|
+---+---+---+---+---+---+

<\ . a
+---+---+---+---+---+---+
|1 2 3 4 5 6|2 3 4 5 6|3 4 5 6|4 5 6|5 6|6|
+---+---+---+---+---+---+
```

The oblique adverb `/.` partitions a *table* along diagonal lines. Thus:

```
</. t=: 1 2 1 */ 1 3 3 1
+---+---+---+---+
|1|3 2|3 6 1|1 6 3|2 3|1|
+---+---+---+---+
```

```

t ; (sum/. t) ; (10 #. sum/. t) ; (121*1331)
+-----+
| 1 3 3 1 | 1 5 10 10 5 1 | 161051 | 161051 |
| 2 6 6 2 |                  |         |         |
| 1 3 3 1 |                  |         |         |
+-----+

```

### Exercises

- 14.1 Define programs analogous to `sum=:+/\` for progressive products, progressive maxima, and progressive minima.
- 14.2 Treat the following programs and comments like those of Section 12, that is, as exercises in reading and writing. Experiment with expressions such as `c pol x` and `c pp d` and `(c pp d) pol x` with `c=:1 3 3 1` and `d=:1 2 1` and `x=:i.5`. See the dictionary or Section 20 for the use of rank:

```

pol=: +/ @ ([*]^i. @# @ [ ]) "1 0      Polynomial
pp=: +//. @ (*/)                      Polynomial product
pp11=: 1 1 & pp                        Polynomial product by 1 1
pp11 d
pp11^:5 (1)
ps=: +/ @ , :                          Polynomial sum

```

## 15. Defined Adverbs

Names may be assigned to adverbs, as they are to nouns and verbs:

```
a=:1 2 3 4 5
prefix=: \
< prefix 'abcdefg'
+-----+
|a|ab|abc|abcd|abcde|abcdef|abcdefg|
+-----+

+ / prefix a
1 3 6 10 15
```

Moreover, new adverbs result from a string of adverbs (such as `/\`) and from a conjunction together with one of its arguments, as well as from other trains listed in the dictionary (II F). Such adverbs can be *defined* by assigning names. Thus:

```
IP=: /\                               Insert Prefix
+ IP a
1 3 6 10 15

with3=: &3
% with3 a
0.333333 0.666667 1 1.33333 1.66667

^ with3 a
1 8 27 64 125

I=: ^: _1                             Inverse adverb
*: I a
1 1.41421 1.73205 2 2.23607

+ IP I 1 3 6 10 15
1 2 3 4 5

ten=: 10&
^. ten 5 10 20 100
0.69897 1 1.30103 2

#. ten 3 6 5
365

from=: -- [. into=: %~
10 into 17 18 19
1.7 1.8 1.9
```

```

10 from 17 18 19
7 8 9

```

```

i=: "_1
{. i i. 3 4
0 4 8

```

Apply to items

### Exercises

- 15.1 Experiment with, and explain the behaviour of, the adverbs

`pow=: ^&` and `log=: &^`.

- 15.2 State the significance of the following expressions, and test your conclusions by entering them:

`+/~ i=: i. 6`

Addition table

`ft=: /~`

Function table adverb

`+ ft i`

Addition table

`! ft i`

Binomial coefficients

`inv=: ^:_1`

Inverse adverb

`sub3=: 3&+ inv`

Subtract-three function

`sub3 i`

## 16. Word Formation

The interpretation of a written English sentence begins with word formation. The process is based on spaces to separate the sentence into units, but is complicated by matters such as apostrophes and punctuation marks: *was* and *Brown* and *Ross*' are each single units, but *however*, is not (since the comma is a separate unit).

The following lists of characters represent sentences in **J**, and can be executed by applying the *do* or *execute* function " . :

```
m=: '3 %: y.'
```

```
d=: 'x.%: y.'
```

```
x.=: 4
```

```
y.=: 27 4096
```

```
" . m
```

```
3 16
```

```
do=: " .
```

```
do d
```

```
2.27951 8
```

The word formation rules of **J** are prescribed in Part I of the dictionary. Moreover, the word-formation function ;: can be applied to the string representing a sentence to produce a boxed list of its words:

```
;: m
```

```
+---+---+---+
```

```
|3| %: |y.|
```

```
+---+---+---+
```

```
words=: ;:
```

```
words d
```

```
+---+---+---+
```

```
|x.| %: |y.|
```

```
+---+---+---+
```

The rhematic rules of **J** apply reasonably well to English phrases:

```
words p=: 'Nobly, nobly, Cape St. Vincent'
```

```
+-----+-----+-----+-----+-----+
```

```
|Nobly| , |nobly| , |Cape|St.|Vincent|
```

```
+-----+-----+-----+-----+-----+
```

```
>words p
```

```
Nobly
```

```
,
```

```
nobly
```

```
,
```

```
Cape
```

```
St.
```

Vincent

### Exercises

- 16.1 Choose sentences such as `pp=: +// .@ ( * / )` from earlier exercises, enclose them in quotes, and observe the effects of word-formation `( ; : )` on them.
- 16.2 The following facility is available under Windows and Macintosh: Move the cursor to the left of a line so that it is separated from the line by one or more spaces, and press Ctrl F1 to display the individually boxed words in the sentence.

## 17. Names and Displays

In addition to the normal names used thus far, there are three further classes:

- 1)  $\$$  : is used for self-reference, allowing a verb to be defined recursively without necessarily assigning a name to it, as illustrated in Section 22.
- 2) The names  $x.$  and  $y.$  are used in explicit definition, discussed in Section 18. They denote the arguments used in explicit definition.
- 3) A name (such as  $ab\_cd\_$ ) that has two underbars of which one is the final character, is a *locative*. Names used in a *locale*  $F$  can be referred to in another locale  $G$  by using the suffix  $F$  in a locative name of the form  $pqr\_F\_$ , thus avoiding conflict with otherwise identical names in the locale  $G$ . See Section I of Part II for further details.

The form of the display invoked by entering a name alone is established by  $9! : 3$ , as described in the appendix. For example:

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

```
9! : 3 (4)           Tree form
```

```
mean
+- / --- +
--+- %
+- #
```

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

```
mean
+/ % #
```

Multiple displays are also possible:

```
9! : 3 (5 4 2)
```

```
mean
+/ % #
+- / --- +
--+- %
+- #
+-----+--+
|+-+--+| % |#|
|+|/||
|+-+--+|
```



+-----+--++

### Exercises

17.1 Experiment with the use of locatives.

## 18. Explicit Definition

The sentences in the example:

```
m=: '3 %: y.'
d=: 'x. %: y.'
```

that were executed in the discussion of word formation, can be used with the *explicit definition* conjunction `: to produce a verb:`

```
script=: 0 : 0
3 %: y.
:
x. %: y.
)
roots=: 3 : script
roots 27 4096
3 16
4 roots 27 4096
2.27951 8
```

Adverbs and conjunctions may also be defined explicitly. For example:

```
table=: 1 : 0
[ by ] over x./          Verbs by and over from Section 3
)
```

```
2 3 5 ^ table 0 1 2 3
+---+
| 0 1 2 3 |
+---+
| 2 | 1 2 4 8 |
| 3 | 1 3 9 27 |
| 5 | 1 5 25 125 |
+---+
```

Control structures may also be used. For example:

<pre>f=: 3 : 0 if. y.&lt;0 do. *:y. else. %:y. end. y.=. y.-1 )  f"0 (_4 4) 16 2</pre>	<pre>factorial=: 3 : 0 a=. 1 while. y.&gt;1 do. a=. a*y. end. a )  factorial"0 i. 6 1 1 2 6 24 120</pre>
--	--

### Exercises

- 18.1 Experiment with and display the functions `roots=: 3 : script` and `13 : script` (which are equivalent).
- 18.2 See the discussion of control structures in the dictionary, and use them in defining further verbs.
- 18.3 Experiment with expressions such as `! d b=: i.7`, after defining the adverb `d`:

```
d=: 1 : 0
+:@x.
)
```

- 18.4 Using the program `pol` from Exercise 14.2, perform the following experiments and comment on their results:

```
g=: 11 7 5 3 2 & pol
e=: 11 0 5 0 2 & pol
o=: 0 7 0 3 0 & pol
(g = e + o) b=: i.6
(e = e@-) b
(o = -@o@-) b
```

Answer: The function `g` is the sum of the functions `e` and `o`. Moreover, `e` is an even function (whose graph is reflected in the vertical axis), and `o` is an odd function (reflected in the origin).

- 18.5 Review Section H of Part II and use scripts to make further explicit definitions.
- 18.6 Enter the following explicit definition of the adverb `even` and perform the suggested experiments with it, using the functions defined in the preceding exercise:

```
even=: 2 : 0
-:@(x.f. + x.f.@-)
)
ge=: g even
(e = ge) b
(e = e even) b
```

18.7 Define an adverb `odd` and use it in the following experiments:

```
exp=: ^
sinh=: 5&o.
cosh=: 6&o.
(sinh = exp odd) b
(sinh = exp .: -) b      The primitive odd adverb .: -
(cosh = exp even) b
(exp = exp even + exp odd) b
```

18.8 The following experiments involve complex numbers, and should perhaps be ignored by anyone unfamiliar with them:

```
sin=: 1&o.
cos=: 2&o.
(cos = ^@j. even) b
(j.@sin = ^@j. odd) b
```

## 19. Tacit Equivalents

Verbs may be defined either explicitly or tacitly. In the case of a one-sentence explicit definition, of either a monadic or dyadic case, the corresponding tacit definition may be obtained by using the adverb `13` : as illustrated below. First enter `9! : 3 ] 2 5` so as to obtain both the boxed and linear displays of verbs:

```

      s=: 0 : 0
(+/y.) % (#y.)
)

```

```
mean=: 3 : s
MEAN=: 13 : s
mean
```

```

+--+--+-----+
| 3 | : | (+/y.) % (#y.) |
+--+--+-----+
3  :  ' (+/y.) % (#y.) '

```

MEAN

+	-	-	-	+	+	-	+
	+	-	+	-	+	%	#
	+			/			
	+	-	+	-	+		
+	-	-	-	+	+	-	+
+	/	%	#				

The explicit form of definition is likely to be more familiar to computer programmers than the tacit form. Translations provided by the adverb `13 :` may therefore be helpful in learning tacit programming. An explicit definition of a conjunction may be translated similarly by the adverb `12 :`. For example:

```

t=: 0 : 0
y.^:_1 @ x.&y.
)

```

```
under=: 2 : t
times=: + under ^.
3 times 4
```

12

$$3 + (u =: 12 : t) \wedge . 4$$

12

u

```

+-----+-----+-----+
| +-----+-----+-----+ & ] . |
| | +-----+-----+ | @ [ . | | | | | | |
| | | . | ^ : | _1 | | | | |
| +-----+-----+-----+ | | | |
+-----+-----+-----+
( ( . . ^ : _1 ) @ [ . ) & ] .

```

**Exercises**

- 19.1 Use the display of the tacit definition of `MEAN` to define an equivalent function called `M`.

Answer: `M=: +/ % #`

## 20. Rank

The shape (\$), tally (#), and rank (#@\$), of a noun are illustrated by the noun `report`, which may be construed as a report covering two years of four quarters of three months each:

```

]report=: i. 2 4 3
0 1 2
3 4 5
6 7 8
9 10 11

12 13 14
15 16 17
18 19 20
21 22 23

```

(\$ ; # ; #@\$) report	Shape, Number of items, Rank
+-----+-----+	
2 4 3   2   3	
+-----+-----+	

The last  $k$  axes determine a  $k$ -cell of a noun; the 0-cells of `report` are the atoms (such as 4 and 14), the 1-cells are the three-element quarterly reports, and the two-cells (or *major cells* or *items*) are the two four-by-three yearly reports.

The rank conjunction `"` is used in the phrase `f" k` to apply a function `f` to each of the  $k$ -cells of its argument. For example:

```

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

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

<@i. s=: 2 5
+-----+
| 0 1 2 3 4 |
| 5 6 7 8 9 |
+-----+

<@i. "0 s
+---+-----+
| 0 1 | 0 1 2 3 4 |
+---+-----+

```

Both the left and right ranks of a dyad may be specified. Thus:

```

      10 11 12 ( , "0 1 ; , "1 1 ; , "1) 0 1 2
+-----+-----+-----+
| 10 0 1 2 | 10 11 12 0 1 2 | 10 11 12 0 1 2 |
| 11 0 1 2 |                   |                   |
| 12 0 1 2 |                   |                   |
+-----+-----+-----+

```

The *basic characteristics* adverb `b.` is very useful in analyzing functions (or expressions that define them) with respect to their ranks. For example:

```

      (# b. 0) ; (+/\ b. 0) ; (+/\ % #) b. 0
+-----+-----+-----+
| _ 1 _ | _ 0 _ | _ _ _ |
+-----+-----+-----+

```

### Exercises

- 20.1 Observe the results of the following uses of the monads produced by the rank conjunction, and comment on them:

```

a=: i. 3 4 5
<"0 a
<"1 a
<"2 a
<"3 a
< a
<"_1 a
<"_2 a
mean=: +/ % #
mean a
mean"1 a
mean"2 a

```

Answer: `<"k` applies `<` to each cell of rank `k`, with `<"(#$a)` `a` being equivalent to `<a`. Moreover, a negative value of `k` specifies a *complementary* rank that is effectively `|k|` less than the rank of the argument `a`.



- 20.2 Use the results of the following experiments to state the relation between the conjunctions  $@$  (*Atop*) and  $@:$  (*At*), and compare your conclusions with the dictionary definition:

```
(g=: <"2) a=: i. 3 4 5
|. @: g a
|. @ g a
|: @: (<"1) a
|: @ (<"1) a
```

Answer: The rank of the function  $|. @: g$  is itself infinite and  $|. @: g$  therefore applies to the entire list result of  $g a$ , consequently reversing it. On the other hand, the function  $f @ g$  *inherits* the rank of  $g$ , and  $|. @ g$  therefore applies individually to the atoms produced by  $g$ , producing no effect.

- 20.3 Use the results of the following experiments to comment on the use of the rank conjunction in dyads:

```
b=: 'ABC'
c=: 3 5 $ 'abcdefghijklmno'
c
b,c
b , "0 1 c
b , "1 1 c
b , "1 c
```

## 21. Gerund and Agenda

In English, a *gerund* is a noun that carries the force of a verb, as does the noun *cooking* in the phrase *the art of cooking*; and *agenda* is a list of items for action. The *tie* conjunction ``` applies to two verbs to form a gerund, from which elements can be chosen for execution. Thus, if the agenda conjunction `@.` is applied to a gerund, its verb right argument provides the results that choose the elements. For example:

```

g=: +`^
a=: <
2 a 3
1

2 g@.a 3
8

3 g@.a 2
5

+:`-: `*:`%: @. (4&|@<.)"0 i. 10
0 0.5 4 1.73205 8 2.5 36 2.64575 16 4.5

```

The verb produced by `g@.a` is often called a *case* or *case statement*, since it selects one of the “cases” of the gerund for execution.

The *insert* adverb `/` applies to a gerund in a manner analogous to its application to a verb. For example:

```

c=: 3 [ x=: 4 [ power=: _1
g/ c,x,power
3.25

3+x^_1
3.25

```

The elements of the gerund are repeated as required. For example:

```

+`*/1,x,3,x,3,x,1
125

```

The last sentence above corresponds to Horner’s efficient evaluation of the polynomial with coefficients `1 3 3 1` and argument `x`.



**Exercises**

- 21.1 Define a function `f` such that `(x=: 4) f c=: 1 3 3 1` yields the result used as the argument to `+`*/` in Horner's method.

Answer: `f=: } .@, @, .`

## 22. Recursion

The factorial function `!` is commonly defined by the statement that factorial `n` is `n` times factorial `n-1`, and by the further statement that factorial `0` is `1`. Such a definition is called *recursive*, because the function being defined recurs in its definition.

A case statement can be used to make a recursive definition, the case that employs the function under definition being chosen repeatedly until the terminating case is encountered. For example:

```
factorial=: 1: `[ ]*factorial@<:) @. *
factorial "0 i.6
1 1 2 6 24 120
```

Note that `1:` denotes the constant function whose result is `1`.

In the sentence `(sum=: +/) i.5` the verb defined by the phrase `+/` is assigned a name before being used, but in the sentence `+/ i.5` it is used anonymously. In the definition of `factorial` above, it was essential to assign a name to make it possible to refer to it within the definition. However, the word `$:` provides *self-reference* that permits anonymous recursive definition. For example:

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

In the Tower of Hanoi puzzle, a set of `n` discs (each of a different size) is to be moved from post A to post B using a third post C and under the restriction that a larger disc is never to be placed on a smaller. The following is a recursive definition of the process:

```
h=: b` (p, .q, .r)@.c
c=: 1: < [
b=: 2&,@[ $ ]
p=: <:@[ h 1: A. ]
q=: 1: h ]
r=: <:@[ h 5: A. ]

3 h x=: 'ABC'
AABACCA
BCCBABB

0 1 2 3 4 <@h"0 1 x
++-+-----+
| A | AAC | AABACCA | AACABBAACCBACAAC |
| B | CBB | BCCBABB | CBBCACCBBAABCBB |
++-+-----+
```

### Exercises

22.1 Use the following as exercises in reading and writing:

$f = :1 : \backslash ( + / / . @ ( , : \sim ) @ ( \$ : @ < : ) ) @ . *$	Binomial Coeffs
$< @ f " 0 \ i . 6$	Boxed binomials
$g = :1 : \backslash ( ( [ , + / @ ( \_ 2 \& \{ . ) ) @ \$ : @ < : ) @ . *$	Fibonacci

## 23. Iteration

The repetition of a process, or of a sequence of similar processes, is called *iteration*. Much iteration is implicit, as in `*/b` and `a*/b`, and `a*b`; explicit iteration is provided by the power conjunction (`^:`), by agenda (`@.`) with self-reference, and by control structures:

```
(cos=: 2&o.) ^: (i.6) b=: 1
1 0.540302 0.857553 0.65429 0.79348 0.701369
    ]y=: cos^:_ b
0.739085
    y=:cos y
1
```

The example `cos^:_` illustrates the fact that infinite iteration is meaningful (that is, terminates) if the process being applied converges.

*Controlled* iteration of a process `p` is provided by `p^:q`, where the result of `q` determines the number of applications of `p` performed. The forms `$:@p^:q` and `p^:q^:_` are commonly useful.

If `f` is a continuous function, and if `f i` and `f j` differ in sign, then there must be a *root* `r` between `i` and `j` such that `f r` is zero; the list `b=:i,j` is said to *bracket* a root. A narrower bracket is provided by the mean of `b` together with that element of `b` whose result on applying `f` differs in sign from its result. Thus:

```
f=: %: - 4:                A sample function

root=: 3 : 0
m=: +/ % #
while. ~:/y.
do.
    if. ~:/ * f ({.,m) y.
        do. y.=. ({.,m) y.
        else. y.=. ({:,m) y.
        end.
    end. m y.
)

b=: 1 32
root b
16

f b,root b
_3 1.65685 1.77636e_15
```

**Exercises**

- 23.1 Use the function `root` to find the roots of various functions, such as `f=: 6&-@!`
- 23.2 Experiment with the function `fn=: +/\` (which produces the figurate numbers when applied repeatedly to `i.n`), and explain the behaviour of the function `fn^:(?@3:)`



## 24. Trains

The train of nouns in the English phrase *Ontario museum Egyptian collection* represents a single noun. Similarly, the fork discussed in Section 5 and its exercises permit the use of arbitrarily long trains of verbs to produce a verb.

Section 15 introduced the use of trains of adverbs, and of conjunctions together with nouns or verbs, to represent adverbs. *Conjunctions* may also be produced by trains of adverbs and conjunctions in a manner analogous to forks.

For example, the case diagrammed on the right below can be used as follows:

```

cj=: \@ \
< cj (+/) a=: i.3 3
+-----+
| 0 1 2 | 0 1 2 | 0 1 2 |
|       | 3 5 7 | 3 5 7 |
|       |       | 9 12 15 |
+-----+

```

```

c
/ \
a1 a2
|  |
x  y

```

```

(<\)@(+/\) a
+-----+
| 0 1 2 | 0 1 2 | 0 1 2 |
|       | 3 5 7 | 3 5 7 |
|       |       | 9 12 15 |
+-----+

```

```

(*/) cj (+/) a
0 1 2
0 5 14
0 60 210

```

The explicit form of defining conjunctions treated in the exercises of Section 18 can be used to produce an equivalent conjunction CJ as shown below. The corresponding tacit definition is produced by 12 : s :

```

s=: 0 : 0
(x.\)@(y.\)
)

CJ=: 2 : s
(*/) CJ (+/) a
0 1 2
0 5 14
0 60 210

12 : s
\ @ \

```

### Exercises

- 24.1 Use the display of 12 : s as a guide in defining an equivalent conjunction C, and compare the resulting definition with the simpler definition used for cj.

## 25. Permutations

Anagrams are familiar examples of the important notion of *permutations*:

```
w=: 'STOP'
3 2 0 1 { w
POST
```

```
2 3 1 0 { w
OPTS
```

```
3 0 2 1 { w
PSOT
```

The left arguments of `{` above are themselves permutations of the list `i.4`; examples of *permutation vectors*, used to represent permutation functions in the form `p&{`.

If `p` is a permutation vector, the phrase `p&C.` also represents the permutation `p&{`. However, other cases of the *cycle* function `C.` are distinct from the *from* function `{`. In particular, `C. p` yields the *cycle representation* of the permutation `p`. For example:

```
]c=: C. p=: 2 4 0 1 3
+---+-----+
|2 0|4 3 1|
+---+-----+
      c C. 'ABCDE'
CEABD
      C. c
      2 4 0 1 3
```

Each of the boxed elements of a cycle specify a list of positions that cycle among themselves; in the example above, the element from position 3 moves to position 4, element 1 moves to 3, and element 4 to 1.

A permutation can be identified by its index in the table of all  $n!$  permutations of order  $n$  listed in increasing order. This index is the *anagram index* of the permutation; the corresponding permutation is effected by the function `A.` as illustrated below:

```
1 A. 'ABCDE'
ABCED
      A. 0 1 2 4 3
      1

(i.!3) A. i.3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
      (i.!3) A. 'ABC'
      ABC
      ACB
      BAC
      BCA
      CAB
```

2 1 0

CBA

**Exercises**

- 25.1 Use the following as exercises in reading and writing (try on `a=: 'abcdef'` and `b=: i. 6` and `c=: i. 6 6`):

<code>f=: 1&amp;A.</code>	Interchange last two items
<code>g=: 3&amp;A.</code>	Rotate last three items
<code>h=: 5&amp;A.</code>	Reverse last three items
<code>i=: &lt;:@!@[ A. ]</code>	<code>k i a</code> reverses last <code>k</code> items

- 25.2 Experiment with the following expressions and others like them to determine the rules for using abbreviated arguments to `C.` and compare your conclusions with the dictionary definitions:

```
2 1 4 C. b=:i.6
(<2 1 4) C. b
(3 1;5 0) C. b
```

- 25.3 Make a program `ac` that produces a table of the cycle representations of all permutations of the order of its argument, as in `ac 3`.

Answer: `ac=: C.@(i.@! A. i.)`

## 26. Linear Functions

A function  $f$  is said to be *linear* if  $f(x+y)$  equals  $(f\ x)+(f\ y)$  for all arguments  $x$  and  $y$ . For example:

```
f=:3&|. @+:@|.
|x=: i.# y=:2 3 5 7 11
0 1 2 3 4
  x+y          f x+y
2 4 7 10 15    8 4 30 20 14
(f x),:(f y)    (f x)+(f y)
2 0 8 6 4       8 4 30 20 14
6 4 22 14 10
```

A linear function can be defined equivalently as follows:  $f$  is linear if  $f@:+$  and  $+\&f$  are equivalent. For example:

```
x f@:+ y      x +&f y
8 4 30 20 14   8 4 30 20 14
```

If  $f$  is a linear function, then  $f\ y$  can be expressed as the *matrix product*  $mp\&M\ y$ , where

```
mp=: +/ . *
M=: f I=: = i.#y      I is an identity matrix

mp&M y      f y
6 4 22 14 10  6 4 22 14 10
```

Conversely, if  $m$  is any square matrix of order  $\#y$ , then  $m\&mp$  is a linear function on  $y$ , and if  $m$  is invertible, then  $(\%.m)\&mp$  is its inverse:

```
x=: 1 2 3 [ y=: 2 3 5
]m=: ? 3 3$9
5 7 3
7 2 3
4 4 2

]n=: %.m
_1.33333 _0.333333 2.5
_0.333333 _0.333333 1
3.33333 1.33333 _6.5
```

```

      g=: mp&m
      h=: mp&n
x g@:+ y      x +&g y
82 63 40      82 63 40

      g h y
2 3 5

```

### Exercises

- 26.1 For each of the following functions, determine the matrix  $M$  such that  $M (mp=: +/ . *) N$  is equivalent to the result of the function applied to the matrix  $N$ , and test it for the case  $N=: i. 6 6$ :

```

| .
-
+ :
( 4&*-2&*@| . )
2&A.

```

## 27. Obverse and Under

The result of  $f^{\wedge} \_1$  is called the *obverse* of the function  $f$ ; if  $f =: g \ . \ h$ , this obverse is  $h$ , and it is otherwise an inverse of  $f$ . Inverses are provided for over 25 primitives (including the case of the square root illustrated in Section 11), as well as many bonded dyads such as  $\neg 3$  and  $10 \&^{\wedge}.$  and  $2 \& \circ ..$  Moreover,  $u @ v^{\wedge} \_1$  is given by  $(v^{\wedge} \_1) @ (u^{\wedge} \_1)$ . For example:

```
fFc=: (32&+ )@(*&1.8)
]b=:fFc _40 0 100
_40 32 212

cFf=: fFc^:_1
cFf b
_40 0 100
```

The result of the phrase  $f \& . g$  is the verb  $(g^{\wedge} \_1) @ (f \& g)$ . The function  $g$  can be viewed as *preparation* (which is done before and undone after) for the application of the “main” function  $f$ . For example:

```
b=: 0 0 1 0 1 0 1 1 0 0 0
sup=: </\                               Suppress ones after the first
sup b
0 0 1 0 0 0 0 0 0 0 0

|. sup |. b                               Suppress ones before the last
0 0 0 0 0 0 0 0 1 0 0 0

sup&.|. b
0 0 0 0 0 0 0 0 1 0 0 0

3 +&.^ . 4                               Multiply by applying the exponential
12                                         to the sum of logarithms

(^.3)+(^.4)
2.48491

^ (^ .3)+(^.4)
12

]c=: 1 2 3;4 5;6 7 8
+-----+-----+
|1 2 3|4 5|6 7 8|
+-----+-----+

|. & . > c                               Open, reverse, and then box

+-----+-----+
|3 2 1|5 4|8 7 6|
+-----+-----+
```

### Exercises

27.1 Use the following as exercises in reading and writing. Try using arguments such as  
`a=: 2 3 5 7` and `b=: 1 2 3 4` and `c=: <@i."0 i. 3 4:`

<code>f=: +&amp;.^</code>	Multiplication by addition of natural logs
<code>g=: +&amp;.(10&amp;^.)</code>	Multiplication using base-10 logs
<code>h=: *&amp;.^</code>	Addition from multiplication
<code>i=:  .&amp;.&gt;</code>	Reverse each box
<code>j=: +/&amp;.&gt;</code>	Sum each box
<code>k=: +/&amp;&gt;</code>	Sum each box and leave open

## 28. Identity Functions and Neutrals

The monads `0&+` and `1&*` are *identity* functions, and `0` and `1` are said to be *identity elements* or *neutrals* of the dyads `+` and `*` respectively. Insertion on an empty list yields the neutral of the dyad inserted. For example:

<code>0</code>	<code>+/ i.0</code>	<code>0</code>	<code>+/ ''</code>	<code>0</code>	<code>+/0{. 2 3 5</code>
<code>1</code>	<code>*/i.0</code>	<code>1</code>	<code>*/''</code>	<code>1</code>	<code>*/0{. 2 3 5</code>

These results are useful in partitioning lists; they ensure that certain obvious relations continue to hold even when one of the partitions is empty. For example:

```

+/ a=: 2 3 5 7 11
28
(+/4{.a)+(+/4}.a)
28
(+/0{.a)+(+/0}.a)
28
*/a
2310
(*/4{.a)*(*/4}.a)
2310
(*/0{.a)*(*/0}.a)
2310

```

The identity functions and other basic characteristics of functions (such as *rank*) are given by the adverb `b.`, as illustrated below:

<code>^ b. _1</code>	Inverse
<code>^.</code>	
<code>^ b. 0</code>	Ranks
<code>_ 0 0</code>	
<code>^ b. 1</code>	Identity function
<code>\$&amp;1@({}.@\$)</code>	



## Exercises

28.1 Predict and test the results of the following expressions:

```
*/ ''
<./ ''
>./ ''
>./0 4 4 $ 0
+/. */ 0 4 4 $ 0
1 2 3 4 +&.^./ 5 6 7 8
```

28.2 Experiment with the dyad `{@;` and give the term used to describe it in mathematics.

Answer: Cartesian product

28.3 Test the assertion that the monads `(%:@~. +/ . * =)` and `%:` are equivalent, and state the utility of the former when applied to a list such as `1 4 1 4 2` that has repeated elements.

Answer: The function `%:` (which could be a function costly to execute) is applied only to the distinct elements the argument (as selected by the *nub* function `~.`).

28.4 Comment on the following experiments before reading the comments on the right:

<code>a=: 2 3 5 [ b=: 1 2 4</code>	
<code>a (f=: *:@+) b</code>	Square of sum
<code>a (g=: +&amp;*: + +:@*) b</code>	Sum of squares plus double product
<code>a (f=g) b</code>	Expression of the identity of the functions
<code>a (f=:g) b</code>	<code>f</code> and <code>g</code> in a <i>tautology</i> (whose result is
<code>taut=: f=:g</code>	<i>always</i> true; that is, <code>1</code> ).

28.5 A phrase such as `f=:g` may be a tautology for the dyadic case only, for the monadic case only, or for both. Use the following tautologies as reading and writing exercises, including statements of applicability (Dyad only, etc.):

<code>t1=: &gt;: -: &gt; +. =</code>	(Dyad only) The primitive <code>&gt;:</code> is identical to greater than <i>or</i> equal
<code>t2=: &lt;. -: -@&gt;.&amp;-</code>	(Both) Lesser-of is neg on greater-of on neg; Floor is neg on ceiling on neg
<code>t3=: &lt;. -: &gt;.&amp;.-</code>	Same as <code>t2</code> but uses <i>under</i>

t4=: *: @>: -: *: + +: + 1:	(Monad) Square of a+1 is square of a plus twice a plus 1
t5=: *: @>: -: #.&1 2 1"0	Same as t4 using polynomial
t6=: ^&3 @>: -: #.&1 3 3 1"0	Like t5 for cube
bc=: i. @>: ! ]	Binomial coefficients
t7=: (>: @)^[ ] -: ( ]# . bc@[ ]"0	Like t6 with k&t7 for kth power
s=: 1&o. [ . c=: 2&o.	Sine and Cosine
t8=: s@+ -: (s@[ *c@]) + (c@[ *s@])	(Dyad) Addition and Subtraction
t9=: s@- -: (s@[ *c@]) - (c@[ *s@])	Formulas for sine
det=: -/ . *	Determinant
perm=: +/ . *	Permanent
sct=: 1 2&o."0@ ( , "0)	Sine and cosine tables
t10=: s@- -: det@sct	Same as t9 but using the determinant of the sin and cos table
t11=: s@+ -: perm@sct	Like t8 using the permanent
S=: 5&o. [ . C=: 6&o.	Hyperbolic sine and cosine
SCT=: 5 6&o."0@ ( , "0)	Sinh and Cosh table
t12=: S@+ -: perm@SCT	Addition theorem for sinh
SINH=: ^ . -: -	Odd part of exponential
COSH=: ^ . . -: -	Even part of exponential
t13=: SINH -: S	Sinh is odd part of exponential
t14=: COSH -: C	Cosh is the even part of exponential
sine=: ^&.j. . -: -	Sine is the odd part of exponential
t15=: sine -: s	under multiplication by 0j1

28.6 Comment on the following expressions before reading the comments on the right:

`g=: + > > .`

Test if sum exceeds maximum

`5 g 2`

True for positive arguments

`5 g _2 _1 0 1 2`

but not true in general

`f=: *.&(0<)`

Test if both arguments exceed 0

`theorem=: f <: g`

The truth value of the result of `f` does not exceed that of `g`. This may also be stated as “if `f` (is true) then `g` (is true)” or as “`f` implies `g`”

`5 theorem _2 _1 0 1 2`

## 29. Secondaries

It is convenient to supplement the *primitives* or *primaries* provided in a language by *secondaries* whose names belong to an easily recognized class. The following examples use names beginning with a capital letter:

Ad=:	[ 0: } -@> : @\$@ ] { . ]	Append diagonal scalar
Ai=:	> : @i .	Augmented integers
Area=:	[ : Det ] , . % @ ! @ # " 1	Area (Vol) try Area tet=:0 , =i . 3
Bc=:	i . ! / i .	Binomial coefficients
Bca=:	% . @Bc	Binomial coefficients (alternating)
By=:	' ' & i @ , . @ [ , . ]	By (format; see Ta)
Cpa=:	] % . i . @ # @ ] ^ / Ei @ [	Coeffs of poly approx
CPA=:	( @ ] ) % . i . @ # @ ] ^ / Ei @ [	Coeffs of poly approx (adverb)
Det=:	- / . *	Determinant
Dpc=:	1 : } . ] * i . @ #	Differentiate poly coeffs
Dl=:	( " 0 ) ( D . 1 )	Derivative (scalar, first)
Ei=:	i . @ ( + * + 0 & = )	Extended integers
Epc=:	Bc @ # X ]	Expand poly coeffs
Ipc=:	0 : , ] % Ai @ #	Integrate poly coeffs
Inv=:	^ : _ 1	Inverse
Id=:	= @ i .	Identity matrix
Mat=:	- : / : ~	Monotone ascending test
Mdt=:	- : \ : ~	Monotone descending test
Mrg=:	+ & \$ { . , @ (   : @ , : )	Merge
Over=:	( { . i } . ) @ " : @ ,	Over (format; see Ta)
Pad=:	2 : ' x . % . ] ^ / Ei @ ( y . " _ ) '	Polynomial approx of degree
Pp=:	+ / / . @ ( * / )	Polynomial coeffs product
Si=:	( Ei @ + : -   ) : ( - / i . )	Symmetric and subsiding int
Span=:	2 : ' y . " _ x . \ ] '	Span of apply of left arg
S1=:	: @   @ ( ^ ! . _ 1 / ~ % . ^ / ~ ) @ i .	Stirling numbers (1st kind)
S2=:	: @ ( ^ / ~ % . ^ ! . _ 1 / ~ ) @ i .	Stirling numbers (2nd kind)
Ta=:	1 : ' [ By ] Over x . / '	Table adverb
Thr=:	] * 0 . 1 & ^ @ [ < :   @ ]	Threshold for non-zero
Tile=:	\$ @ ] { . [ \$ ~ \$ @ ] + 2 :   1 : + \$ @ ]	Tile (try 0 1 Tile i . 2 3 4)
X=:	+ / . *	Times (matrix product)
XA=:	- / . *	Times (alternating)

### Exercises

- 29.1 Enter the definitions of the secondaries (or at least those used in these exercises), and then enter the following expressions:

```
(Ai 2 3);(Ai 2 _3);(Ei 2 3);(Ei 2 _3)

(i.;i.@-;Ai;Ai@-;Ei;Ei@-) 4

(Si 4);(7 4 Si 4)

+Ta~@i. 4

(S1;S2) 7

(];X/;%/;%./;(%./%{.)) y=: (Bc ,: Bca) 5

(0 1&Cb;1 _1&Cb) i. 2 3 4
```

- 29.2 Perform further experiments with the secondaries.



# Sample Topics

This part provides examples of the use of **J** in various topics; it is designed to be used in conjunction with the dictionary and at the keyboard of a **J** system. It is also designed to be used *inductively*, as follows:

- Read one or two sentences and their results (which begin at the left margin), and attempt to state clearly in English what each sentence does.
- Enter similar sentences to test the validity of your statements.
- Consult the dictionary to confirm your understanding of the meaning of primitives such as `i.` (used with one argument or two). Use the Vocabulary at the end of the book as an index to pages in the dictionary. In the Windows system, highlight a word (e.g. `/:`) and hit ctrl-F1 to display the dictionary entry for that word.
- Enter *parts* of a complex sentence, such as `i. 26` and `j+/i.26` in the case of `a.{~j+/i.26` used on the following page.

## 1. Spelling

```
phr=: 'index=: a.i. 'aA' '
;:phr
+-----+-----+-----+
|index|=: |a. |i. | 'aA' |
+-----+-----+-----+

$ ;:phr
5

> ;:phr
index
=:
a.
i.
'aA'

(do=: ".) phr
97 65

do 'abc =: 3 1 4 2'
3 1 4 2

abc
3 1 4 2
```

2. Alphabet and Numbers

(a. {~ j +/ i.26) ; (j +/ i.6) ; (j=: a. i. 'aA') ; (\$ a.)

abcdefghijklmnopqrstuvwxyz	97 98 99 100 101 102	97 65	256
ABCDEFGHIJKLMNOPQRSTUVWXYZ	65 66 67 68 69 70		

1 2 3{ t=: 8 32\$a.  
!"#\$%&'()\*+,-./0123456789:;<=>?  
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_  
`abcdefghijklmnopqrstuvwxyz{|}~•

The major alphabet

The table `t` arranges the alphabet in eight rows, but its complete display would look odd because of the effects of various control characters such as the carriage return. A boxed display `<t` is more readable because spaces are substituted for them.

i. 2 5  
0 1 2 3 4  
5 6 7 8 9

Table of integers

r=: 0j1 \_1 0j\_1 1  
+ r  
0j\_1 \_1 0j1 1

Square roots of plus and minus 1  
(Complex) conjugates

r \* +r  
1 1 1 1

r \*/ r  
\_1 0j\_1 1 0j1  
0j\_1 \_1 0j1 \_1  
1 0j1 \_1 0j\_1  
0j1 \_1 0j\_1 1

Multiplication table of roots of unity

! 45x  
119622220865480194561963161495657715064383733760000000000

“x” denotes extended precision

3. Grammar

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) i. 9
0 1 8 27 64 125 216 343 512

```

### Parts Of Speech

50 fahrenheit	Nouns/Pronouns
+ - * % cube	Proverbs
/ \	Adverbs
&	Conjunction
=:	Copula
( )	Punctuation

## 4. Function Tables

Just as the behaviour of *addition* is made clear by addition tables in elementary school, so the behaviour of other verbs (or *functions*) can be made clear by function tables.

The next few pages show how to make function tables, and how to use the *utility* functions `over` and `by` to border them with their arguments to make them easier to interpret.

Study the tables shown, and make tables for other functions (such as `<`, `<.` and `%`) suggested by the Vocabulary.

<pre>       (+/~ ; */~) 0 1 2 +-----+-----+   0 1 2   0 0 0     1 2 3   0 1 2     2 3 4   0 2 4   +-----+-----+ </pre>	Addition and Times tables
<pre>       ^/ ~ i. 4 1 0 0 0 1 1 1 1 1 2 4 8 1 3 9 27 </pre>	Power table

```
+. / ~ 0 1
0 1
1 1
```

Or table

5. Bordering a Table

```
over=: ({. ;}. )@":@,
by=: ' '&i@, .@[ ,. ]
primes=: 2 3 5 [ i=: 0 1 2 3 4
primes by i over primes */ i
```

Utility functions, intended for use rather than for immediate study

```
+--+-----+
| |0 1  2  3  4|
+--+-----+
|2|0 2  4  6  8|
|3|0 3  6  9 12|
|5|0 5 10 15 20|
+--+-----+
```

```
tba=: 1 : '[ by ] over x. / ' I
primes * tba
```

Table adverb

```
+--+-----+
| |0 1  2  3  4|
+--+-----+
|2|0 2  4  6  8|
|3|0 3  6  9 12|
|5|0 5 10 15 20|
+--+-----+
```

```
7 11 ^ tba i
+--+-----+
| |0 1  2  3  4|
+--+-----+
| 7|1  7  49 343 2401|
|11|1 11 121 1331 14641|
+--+-----+
```

## 6. Tables (Letter Frequency)

```

text=: ' i sing of olaf glad and big'
alph=: ' abcdefghijklmnopqrstuvwxyz'
10{.alph=/text
1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

'01'{~10{.alph=/text
1010000100100001000010001000
0000000000000010000100100000
0000000000000000000000000100
0000000000000000000000000000
00000000000000000000100010000
0000000000000000000000000000
0000000010000100000000000000
0000001000000000100000000001
0000000000000000000000000000
0100100000000000000000000010

]LF=: 2 13 $ +/"1 alph =/ text
7 3 1 0 2 0 2 3 0 3 0 0 2
0 2 2 0 0 0 1 0 0 0 0 0 0

+ / + / LF
28
$text
28

```

Letter frequency table

## 7. Tables

```
div=: 0=rem=: i | /i=: i.7
(,i) ; rem ; div
```

Divisibility and remainder tables

0	0	1	2	3	4	5	6	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
3	0	1	2	0	1	2	0	1	0	0	1	0	0	1	0
4	0	1	2	3	0	1	2	1	0	0	0	1	0	0	0
5	0	1	2	3	4	0	1	1	0	0	0	0	1	0	0
6	0	1	2	3	4	5	0	1	0	0	0	0	0	1	0

```
(i#~2=+/div) ; (2=+/div) ; (+/div)
```

Primes, test, # of divisors

2	3	5	0	0	1	1	0	1	0	7	1	2	2	3	2	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
(= /~ ; < /~ ; < : /~ ; ~ : /~) i. 5
```

1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1

```
t=: (/ ~) (@i.)
```

Table adverb

```
(=t ; <t ; <:t ; ~:t) 5
```

1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1

```
(^t ; !t ; -t) 5
```

1	0	0	0	0	0	1	1	1	1	1	0	-1	-2	-3	-4
1	1	1	1	1	1	0	1	2	3	4	1	0	-1	-2	-3
1	2	4	8	16	0	0	1	3	6	2	1	0	-1	-2	0
1	3	9	27	81	0	0	0	1	4	3	2	1	0	-1	0
1	4	16	64	256	0	0	0	0	1	4	3	2	1	0	0

## 8. Classification

Classification is a familiar notion. For example, the classification of letters of the alphabet as *vowel*, *consonant*, *sibilant*, or *plosive*; of colors as *primary* and *secondary*; and of numbers as *odd*, *even*, *prime*, and *complex*.

It is also very important; it provides the basis for many significant notions, such as graphs, barcharts, and sets.

A classification may be *complete*, (each object falling into at least one class), and it may be *disjoint* (each object falling into at most one class). A *graph* is a *disjoint* classification corresponding to the non-disjoint classification used to produce a barchart.

```
x=: 1 2 3 4 5 6 7
  y=: (x-3) * (x-5)
8 3 0 _1 0 3 8
```

Parabola (roots at 3 and 5)

```
range=: >./ - i.@spread
spread=: 1: + >./ - <./
spread y
10
```

```
range y
8 7 6 5 4 3 2 1 0 _1
```

```
((range <:/ ]) ; {&' *'@(range<:/)) y
```

Barcharts of y

```
+-----+-----+
| 1 0 0 0 0 0 0 1 | *      * |
| 1 0 0 0 0 0 0 1 | *      * |
| 1 0 0 0 0 0 0 1 | *      * |
| 1 0 0 0 0 0 0 1 | *      * |
| 1 0 0 0 0 0 0 1 | *      * |
| 1 1 0 0 0 0 1 1 | **     ** |
| 1 1 0 0 0 0 1 1 | **     ** |
| 1 1 0 0 0 0 1 1 | **     ** |
| 1 1 1 0 1 1 1 1 | ***    *** |
| 1 1 1 1 1 1 1 1 | ***** |
+-----+-----+
```

9. Disjoint Classification (Graphs)

If only the final element of a boolean list `b` is non-zero, then (and only then) will the result of `</b` be non-zero. Consequently, `</\` applied to `b` suppresses all ones after the first, and the result therefore represents a disjoint classification. For example:

```
cct=: #:i.@(2: ^ #)
b=: |: cct 2 3 5 7
b
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

</b
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

</\b
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

y=: (x-3) * (x-5) [ x=: 1 2 3 4 5 6 7
range=: >./ - i.@spread
spread=: 1: + >./ - <./
bc=: (range <:/]) y
```

Complete classification table

bc;(</\bc);({&' .\*' </\bc)

1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	*	.....	*
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	.....		
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	.....		
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	.....		
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	.....		
1	1	0	0	0	0	1	1	0	1	0	0	0	1	0	0	.*....*		
1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	.....		
1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	.....		
1	1	1	0	0	1	1	1	0	0	1	0	1	0	0	0	..*.*..		
1	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	...*...		

Barchart and graphs

## 10. Classification (with Selection and Inner Product)

The complete classification table can be used in a variety of ways, including use with various inner products. For example:

```
cct=: #:i.@(2: ^ #)
m=: 2 3 5 ,: 4 2 1
n=: |: cct 0{m
m ; n ; m +/ . * n
```

2	3	5	0	0	0	0	1	1	1	1	0	5	3	8	2	7	5	10
4	2	1	0	0	1	1	0	0	1	1	0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1	0	1									

The pattern of the inner product can be seen more clearly in the following display: the element in a given row and column of the matrix product `p` in the lower right box corresponds to the row of the left argument and the column of the right argument. Thus:

```
('' ; n) ,: (m ; p=: m +/ . * n)
```

			0	0	0	0	1	1	1	1
			0	0	1	1	0	0	1	1
			0	1	0	1	0	1	0	1

2	3	5	0	5	3	8	2	7	5	10
4	2	1	0	1	2	3	4	5	6	7

```
(+/r*c) ; (r*c) ; (r=: 0{m) ; (c=: 3{"1 n) ; (<0 3){p
```

8	0	3	5	2	3	5	0	1	1	8
---	---	---	---	---	---	---	---	---	---	---

Just as the ordinary matrix product yields sums over products, the inner product `*/ . ^` yields products over powers. Hence, `m */ . ^ n` produces products over all possible subsets of the rows of `m`:

```
m */ . ^ n
1 5 3 15 2 10 6 30
1 1 2 2 4 4 8 8
```

Also see the use of the *key* adverb `(/. )` for classification.

## 11. Classification (Sets and Propositions)

The list `-.+./t` appended to any classification table `t` will yield a complete classification table, and the function defined below therefore completes a classification table. The function `tab` ensures that a scalar or vector argument is treated as a one-rowed table

```
c=: complete=: (] , (+./ { . ,: )@: -. @: (+./))@: tab
tab=: ,: ^: (0: > . 2: - # @ $)

c 0 0 1, : 0 1 0          c 1 0 1, : 0 1 0
0 0 1          1 0 1
0 1 0          0 1 0
1 0 0

(c 1 0 1);(c c 1 0 1);(c 0);(c 1)
+-----+-----+-----+
|1 0 1|1 0 1|0 1|
|0 1 0|0 1 0|1|
+-----+-----+-----+
```

A function that yields a single boolean list is called a *proposition*; its result is a one-way classification called a *set*. The classification can, of course, be completed by the complementary set. For example:

```
p1=: 2<: *. <&5          Set defined by interval
p1a=: (2<:]) *. (<5:)   Alternative definition
p2=: = <.               Set of integers
a=: 2 %~ i. 11
(],p1,p1a,p2,(p1+.p2),:(p1*.p2)) a
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5
0 0 0 0 1 1 1 1 1 0
0 0 0 0 1 1 1 1 1 0
1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 1 1 1 1 1
0 0 0 0 1 0 1 0 0 0

list=: 1 : 0          Adverb to list elements of set
x. # ]
)

((p1 list);(p2 list);((p1*.p2)list)) a
+-----+-----+-----+
|2 2.5 3 3.5 4 4.5|0 1 2 3 4 5|2 3 4|
+-----+-----+-----+
```



## 12. Sorting

The sort  $x /: y$  re-orders  $x$  according to the grade of  $y$ , that is,  $/:y$ :

```
x=: 2 7 1 8 [ y=: 1 7 3 2
  (/:y);((/:y){x);(x/:y);(x/:x)
+-----+-----+-----+-----+
|0 3 2 1|2 8 1 7|2 8 1 7|1 2 7 8|
+-----+-----+-----+-----+
```

The grade and sort of literal characters is based upon the ordering of the underlying alphabet  $a..$ . For example, if the name "text" is used for the present sentence (up to and including the colon), then:

```
tdw=: >dwds=: ~. wds=: ;: text
($tdw),($dwds),($wds),($text)
21 9 21 25 103
```

```
]alph=: a. {~ , (i. 26) +/ (a.i.'aA')
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

```
tdw; (tdw /: tdw);(tdw/: alph i. tdw)
+-----+-----+-----+
|For      |"        |and      |
|example  |(        |colon    |
|,         |)        |example  |
|if        |,        |for      |
|the       |For      |For      |
|name      |and      |if       |
|"         |colon    |including|
|text      |example  |is       |
|is        |for      |name     |
|used      |if       |present  |
|for       |including|sentence |
|present   |is       |text     |
|sentence  |name     |then:    |
|(         |present  |the      |
|up        |sentence |to       |
|to        |text     |up       |
|and       |the      |used     |
|including |then:    |,        |
|colon     |to       |"        |
|)         |up       |(        |
|then:     |used     |)        |
+-----+-----+-----+
```

The middle column is the alphabetized table of distinct words, but (because the cases are not interleaved in the alphabet  $a..$ ), the words “for” and “For” are widely separated; they are brought together in the last column by indexing the table by a suitable alphabet.

### 13. Compositions (Based on Conjunctions)

In math, the symbol  $\circ$  is commonly used to produce a function defined as the *composition* of two functions:  $f \circ g$  is defined as  $f(g(y))$ . Normally, such composed functions are only defined to apply to a single scalar argument.

**J** provides compositions effected by five distinct conjunctions, as well as compositions effected by isolated sequences of verbs: hooks and forks, and longer trains formed from them. The five conjunctions are  $\&$ ,  $\&.$ ,  $\&:$ ,  $@$  and  $@:$ , the conjunctions  $@$  and  $@:$  being related in the same manner as  $\&$  and  $\&:$ .

The conjunction  $\&$  is closest to the composition  $\circ$  used in math, being identical to it when used for two scalar (rank zero) functions to produce a function to be applied to a single scalar argument. However, it is also extended in two directions:

1. Applied to one verb and one noun it produces a monadic function illustrated by the cases  $10\&.^.$  (Base ten logarithm) and  $\&^3$  (Cube).
2. Applied to two verbs it produces (in addition to the monadic case used in math) a dyadic case defined by:  $x\ f\&g\ y$  is  $(g\ x)\ f\ (g\ y)$ . For example,  $x\ \%!\ y$  is the quotient of the factorials of  $x$  and  $y$ .

The conjunction  $\&.$  applies only to verbs, and  $f\&.\ g$  is equivalent to  $f\&g$  except that the inverse of  $g$  is applied to the final result. For example:

$$\begin{array}{cc} 3\ +\&.^.\ 4 & 3\ +\&.\&.^.\ 4 \\ 2.48491 & 12 \end{array}$$

For scalar arguments the functions  $f\&:\ g$  and  $f\&g$  are equivalent, but for more general arguments,  $g$  applies to each cell as dictated by its ranks. In the case of  $f\&g$ , the function  $f$  then applies to each result produced; in the case of  $f\&:\ g$  it applies to the overall result of all of the cells. For example:

$$\begin{array}{c} ([\ ]\ i\ \%.\ i\ |:\&\%.\ i\ |:\&:\&\%.)\ i.\ 2\ 2\ 2 \\ +-----+-----+-----+ \\ \left| \begin{array}{cc|cc|cc} 0 & 1 & -1.5 & 0.5 & -1.5 & 1 & -1.5 & -3.5 \\ 2 & 3 & & 1 & 0 & 0.5 & 0 & 1 & 3 \end{array} \right| \\ \left| \begin{array}{cc|cc|cc} 4 & 5 & -3.5 & 2.5 & -3.5 & 3 & 0.5 & 2.5 \\ 6 & 7 & & 3 & -2 & 2.5 & -2 & 0 & -2 \end{array} \right| \\ +-----+-----+-----+ \end{array}$$

The conjunctions @ and & agree in the monadic case, as indicated below for cells  $x$  and  $y$  as dictated by the ranks of  $g$ :

$$\begin{aligned} f \& g \ y &\leftrightarrow f \ g \ y \\ f @ g \ y &\leftrightarrow f \ g \ y \\ x \ f \& g \ y &\leftrightarrow (g \ x) \ f \ (g \ y) \\ x \ f @ g \ y &\leftrightarrow f \ (x \ g \ y) \end{aligned}$$

#### 14. Compositions (Based on Hooks and Forks)

A *verb* is produced by trains of three or two verbs, as defined by the following diagrams:



For example,  $5 (+ * -) 3 \leftrightarrow (5 + 3) * (5 - 3)$ .

The foregoing definition is from Section II F of the dictionary. The following examples concern functions used in statistics:

mean=: +/%#	Mean
norm=: ] - +/ % #	Centered on mean
sqcm=: 2: ^~ ] - +/ % #	Sqr of centred on mean
stdv=: [: mean 2: ^~ ] - mean=: +/%#	Standard Deviation
var=: 2: %: [: mean 2: ^~ ] - mean=: +/%#	Variance

```

, . & . > @ ( ] ; mean ; norm ; sqcm ; stdv ; var ) y=: 2 3 4 5
+---+---+---+---+---+
| 2 | 3.5 | -1.5 | 2.25 | 1.25 | 1.11803 |
| 3 |     | -0.5 | 0.25 |     |     |
| 4 |     |  0.5 | 0.25 |     |     |
| 5 |     |  1.5 | 2.25 |     |     |
+---+---+---+---+---+

```

## 15. Junctions

Four functions are commonly used to join arguments: `;`, `,`, `..` and `,:`. We will illustrate them for the cases of vector and matrix arguments:

```
a=: 'pqr' [ b=: 'PQR'
m=: 3 3$ 'abcdefghi' [ n=: 3 3$ 'ABCDEFGHI'
```

```
a (i i , i ,. i ,:) b
```

pqr	PQR	pP	pqr
qQ	PQR	qQ	PQR
rR		rR	

```
m (i i , i ,. i ,:) n
```

abc	ABC	abc	abcABC	abc
def	DEF	def	defDEF	def
ghi	GHI	ghi	ghiGHI	ghi
		ABC		ABC
		DEF		DEF
		GHI		GHI

```
a (i i , i ,. i ,:) n
```

pqr	ABC	pqr	pABC	pqr
DEF	DEF	qDEF	qDEF	
GHI	GHI	rGHI	rGHI	
				ABC
				DEF
				GHI

```
m (i i , i ,. i ,:) b
```

abc	PQR	abc	abcP	abc
def		def	defQ	def
ghi		ghi	ghiR	ghi
		PQR		
				PQR

## 16. Partitions (Adverbs)

Used monadically, the results of the adverbs `\` and `\.` and `/.` provide *prefix*, *suffix*, and *oblique* partitions. They are commonly applied to arithmetic functions such as summation (`+/`), product over (`*/`), and continued fractions (`((+%) /)`); we will also illustrate them for `box` (`<`), which shows their structure more clearly:

```
a=: 2 3 5 7 11 [ t=: 1 2 1 */ 1 3 3 1
, .&.>((+/\a) ; (+/\.a) ; ((+%) /\a) ; (+//.t);t)
```

2	28	2	1	1	3	3	1
5	26	2.333333	5	2	6	6	2
10	23	2.3125	10	1	3	3	1
17	18	2.31304	10				
28	11	2.31304	5				
			1				

```
<\a
```

2	2	3	2	3	5	2	3	5	7	2	3	5	7	11
---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

```
<\.a
```

2	3	5	7	11	3	5	7	11	5	7	11	7	11	11
---	---	---	---	----	---	---	---	----	---	---	----	---	----	----

```
</.t
```

1	3	2	3	6	1	1	6	3	2	3	1
---	---	---	---	---	---	---	---	---	---	---	---

Used dyadically, they provide *infix*, *outfix*, and *key classification*. For example:

```
3 <\ a
```

2	3	5	3	5	7	5	7	11
---	---	---	---	---	---	---	---	----

```
_3 <\ a
```

2	3	5	7	11
---	---	---	---	----

```
2<\.a
```

5	7	11	2	7	11	2	3	11	2	3	5
---	---	----	---	---	----	---	---	----	---	---	---

```

1 2 3 1 3 *//. a
14 3 55

```

## 17. Partitions (Based on Cut Conjunction)

The left argument of the cut conjunction is a function which is applied to various types of partitions specified by the numeric right argument. We will illustrate its behavior for the fixed function *box*, using the adverb *< ;* :

```

t=: 'When eras die/their legacies/are left to/strange
police/'
;: t
+-----+-----+-----+-----+-----+-----+-----+-----+
|When|eras|die|/|their|legacies|/|are|left|to|/|strange|police|/|
+-----+-----+-----+-----+-----+-----+-----+-----+
cut=: < ;.
_2 cut ;: t
+-----+-----+-----+-----+-----+-----+-----+-----+
|+-----+-----+-----+-----+-----+-----+-----+-----+
||When|eras|die|/|their|legacies|/|are|left|to|/|strange|police|/|
|+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
2 cut ;: t
+-----+-----+-----+-----+-----+-----+-----+-----+
|+-----+-----+-----+-----+-----+-----+-----+-----+
||When|eras|die|/|their|legacies|/|are|left|to|/|strange|police|/|
|+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
(i. 3 2 2);(i. 12 12)
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 1| 0 1 2 3 4 5 6 7 8 9 10 11|
| 2 3| 12 13 14 15 16 17 18 19 20 21 22 23|
|    | 24 25 26 27 28 29 30 31 32 33 34 35|
| 4 5| 36 37 38 39 40 41 42 43 44 45 46 47|
| 6 7| 48 49 50 51 52 53 54 55 56 57 58 59|
|    | 60 61 62 63 64 65 66 67 68 69 70 71|
| 8 9| 72 73 74 75 76 77 78 79 80 81 82 83|
|10 11| 84 85 86 87 88 89 90 91 92 93 94 95|
|    | 96 97 98 99 100 101 102 103 104 105 106 107|
|    | 108 109 110 111 112 113 114 115 116 117 118 119|
|    | 120 121 122 123 124 125 126 127 128 129 130 131|
|    | 132 133 134 135 136 137 138 139 140 141 142 143|
+-----+-----+-----+-----+-----+-----+-----+-----+

(i. 3 2 2) 0 cut i. 12 12
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 2 3| 53 54 55 56 57 58 59|105 106 107|
|13 14 15| 65 66 67 68 69 70 71|117 118 119|
|    | 77 78 79 80 81 82 83|129 130 131|
+-----+-----+-----+-----+-----+-----+-----+-----+

```

	89	90	91	92	93	94	95	141	142	143
	101	102	103	104	105	106	107			
	113	114	115	116	117	118	119			

## 18. Geometry

We will illustrate the topic of coordinate geometry by defining functions for polygons in two dimensions that give the displacements between adjacent vertices, the length of sides, the semiperimeter, and the area based on Heron's formula.

We also present a more general definition for area that not only gives the signed area (positive if the vertices appear in counter-clockwise order), but also applies to polyhedra in higher dimensions (in which case the name *area* might better be replaced by *volume*). This definition is based on the use of the determinant applied to a square matrix obtained by bordering a table of vertices `t` by a final row of the values `%!#t`. Thus:

```
(length=: +/&.*:) 5 12
13
disp=: ] - 1&|. "1
sides=: length@disp
semiper=: -:@(+/@)@sides
HERON=: %:@(*/@)@(semiper - 0: , sides)
area=: -/ . * @ (] , %@!@#)

t=: 0 0 4 ,: 3 4 7
(];(1&|. "1);disp;sides;semiper;HERON;area) t
+-----+-----+-----+-----+-----+-----+-----+
| 0 0 4 | 0 4 0 | 0 4 4 | 0 4 4 | 0.5 3.5 4 | 2.29129 0j0.5 0 | _2 |
| 3 4 7 | 4 7 3 | _1 _3 4 | 1 3 4 |          |          |          |
+-----+-----+-----+-----+-----+-----+-----+

tet1=: 6 0 3 0, 3 6 5 8, : 7 4 0 5
tet2=: 0, . =i. 3 3
tet1;(area tet1);tet2;(area tet2)
+-----+-----+-----+-----+
| 6 0 3 0 | 11.5 | 0 1 0 0 | _0.166667 |
| 3 6 5 8 |      | 0 0 1 0 |          |
| 7 4 0 5 |      | 0 0 0 1 |          |
+-----+-----+-----+-----+
```

19. Symbolic Functions

For any function, a corresponding *symbolic* function can be defined to display the expression rather than evaluate it. For example:

```
minus=: [ , '-'_ , ]
'a' minus 'b'
a-b

list=: 'abcd'
table=: 4 4$'ABCDEFGHJKLMNOP'
minus/list
a-b-c-d

(minus/\list);('01'minus"0/list);(minus//.table);table

+-----+-----+-----+
| a      | 0-a    | A      | ABCD  |
| a-b    | 0-b    | B-E    | EFGH  |
| a-b-c  | 0-c    | C-F-I  | IJKL  |
| a-b-c-d| 0-d    | D-G-J-M| MNOP  |
|         | 1-a    | L-O    |        |
|         | 1-b    | P      |        |
|         | 1-c    |        |        |
|         | 1-d    |        |        |
+-----+-----+-----+

(, .list)=: 4 3 2 1
(" . minus/\list) ,: (-/\4 3 2 1)
4 1 3 2
4 1 3 2

3 (minus/\ ; minus/\.) 'abcdefg'

+-----+-----+
| a-b-c | d-e-f-g |
| b-c-d | a-e-f-g |
| c-d-e | a-b-f-g |
| d-e-f | a-b-c-g |
| e-f-g | a-b-c-d |
+-----+-----+
```



## 20. Directed Graphs

A *directed graph* is a collection of *nodes* with *connections* or *arcs* specified between certain pairs of nodes. It can be used to specify matters such as the precedences in a set of processes (stuffing of envelopes must precede sealing), or the structure of a *tree*.

The connections can be specified by a boolean *connection matrix* instead of by arcs, and the connection matrix can be determined from the list of arcs.

The connection matrix is convenient for determining various properties of the graph, such as the *in-degrees* (number of arcs entering a node), the *out-degrees*, immediate descendants, and the *closure*, or connection to every node reachable through some path. For example:

```

from=: 3 7 2 5 5 7 1 5 5 5 2 6 1 2 3 7 7 4 7 2 7 4
to=: 5 6 0 2 6 2 7 6 0 7 3 3 2 1 7 0 4 2 3 0 0 3

$ arcs=: from,.to
22 2

|: arcs { nodes=: 'ABCDEFGH'          Transposed for display
DHCFFHBFFFCGBCDHHEHCHE
FGACGCHGAHDDCBHAECDAAD

CM=: #. e.~ [: i. [ , [              Connection matrix from arcs
]cm=: (>: >:./,arcs) CM arcs
0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 1
1 1 0 1 0 0 0 0
0 0 0 0 0 1 0 1
0 0 1 1 0 0 0 0
1 0 1 0 0 0 1 1
0 0 0 1 0 0 0 0
1 0 1 1 1 0 1 0

(+ /cm) ; (+ /"1 cm) ; (+ /+ /cm) ; (#arcs) ; (#~.arcs)
+-----+-----+-----+-----+
| 3 1 4 4 1 1 2 3 | 0 2 3 2 2 4 1 5 | 19 | 22 | 19 |
+-----+-----+-----+-----+

The foregoing results are the in, out, and total degrees; followed by the number of arcs, and
the number of distinct arcs. A boolean vector b may be used to represent the nodes, and the
inner product b +./ . *. cm gives the same representation of the nodes reachable from
them. The immediate family (which includes the original points themselves) is therefore
given by the function imfam :

imfam=: [ +. +./ . *.
(b=: 1 0 0 0 0 0 0 1) imfam cm
1 0 1 1 1 0 1 1

```

## 21. Closure

Just as `b imfam cm` produces the immediate family of `b`, so does the phrase `cm imfam cm` produce the immediate families of each of the rows of `cm`. We will, however, use a new sparser connection matrix that will be more instructive, and will use powers of `imfam` to produce families of further generations, including an infinite power to give the *closure* of the connection matrix; that is, the connection matrix for all points reachable by a path of any length:

```
cm=: (i. =/ <:@i.) 8
<"2 cm imfam^:0 1 2 _ cm
```

0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1
0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

The closure of `cm` can therefore be expressed as `cm imfam^:_ cm`, and a monadic closure function can be defined as follows:

```
(closure=: imfam^:_ ~) cm
0 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1
0 0 0 1 1 1 1 1
0 0 0 0 1 1 1 1
0 0 0 0 0 1 1 1
0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
```

The complete definition of the closure function may now be displayed as follows:

```
closure f.
([ +. +./ .*.)^:_~
```

## 22. Distance

The “street” distance between two points will be defined as the sum of the magnitudes of the difference of their coordinates along each axis. Thus:

```

d=: +/@:|@:-"1
p=: 3 5 1 [ q=: 7 4 0
p d q

```

```
table=: #: i. 2^3
      (]; d/~) table
```

Table and distances between each pair of points in it

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

```
g=: [ * [ = d/~@]
(];(d/~);(1&g);(2&g)) table
```

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

```
{&' .+*' &.> ( );(d/~);(1&g);(2&g);(3&g)) table
```

[illegible]

## 23. Polynomials

The monadic function  $M = : 3 : * ] ^ 2 :$  is a multiple of an integral power of its argument, and is called a *monomial*; and a sum of monomials such as  $SM = : (3 : * ] ^ 2 : ) + (2.5 " _ * ] ^ 4 : ) + ( _ 5 " _ * ] ^ 0 : )$  is a *polynomial*.

Any polynomial can be expressed in the *standard* form  $c \& p$ , where  $c$  is a suitable list of *coefficients*, and where  $p = : + / @ ( [ * ] ^ i . @ \# @ [ ] " 1 0$ . For example:

```
SM=: (3:*]^2:)+(2.5"_*]^4:)+(_5"_*]^0:)
p=: +/@[[*]^i.@#[ ]"1 0
c=: _5 0 3 0 2.5
x=: _2 _1 0 1 2
(SM x),(c p x),:(c&p x)
47 0.5 _5 0.5 47
47 0.5 _5 0.5 47
47 0.5 _5 0.5 47
```

The primitive  $p$ . is equivalent to the function  $p$  defined above, and will be used hereafter. The polynomial  $c \& p$ . is very important for a number of reasons, including:

1. It applies to any numeric argument, real or complex (and the parameter  $c$  may also be complex).
2. It can be used to approximate a wide range of functions.
3. It is *closed* under a number of operations; that is, the sum, difference, product, the composition  $@$ , the derivative, and the integral of polynomials are themselves polynomials.
4. The coefficients of the results of each case listed in 3 are easily expressed. For example, if  $\#c$  equals  $\#d$ , then  $c \& p$ . +  $d \& p$ . is equal to  $(c+d) \& p$ . . More generally, it is equal to  $(+ / c , : d) \& p$ . . Thus:

$ps = : + / @ , :$	Polynomial sum
$pd = : - / @ , :$	Polynomial difference
$pp = : + / / . @ ( * / )$	Polynomial product
$D = : d . 1$	Scalar (rank 0) first derivative
$pD = : 1 : \} . ] * i . @ \#$	Polynomial derivative
$pI = : 0 : , ] \% 1 : + i . @ \#$	Polynomial integral

## 24. Polynomials (Continued)

For example:

```

c=: 1 2 1 [ d=: 1 3 3 1
x=: 2 1 0 _1 _2
, .&. > ((c pp d); ((c pp d)&p. x); (c&p.*d&p.)x)
+---+---+---+
| 1 | 243 | 243 |
| 5 | 32  | 32  |
|10 | 1   | 1   |
|10 | 0   | 0   |
| 5 | _1  | _1  |
| 1 |      |      |
+---+---+---+

, .&. > ((d&p.D x); (pD d); ((pD d)&p. x); (pI d); pD pI d)
+---+---+---+---+
| 27 | 3 | 27 | 0 | 1 |
|12 | 6 | 12 | 1 | 3 |
| 3 | 3 | 3  | 1.5 | 3 |
| 0 |   | 0  | 1  | 1 |
| 3 |   | 3  | 0.25 |   |
+---+---+---+---+

le=: c(ps pp pd)d
0 _2 _9 _16 _14 _6 _1

e&p. x
_648 _48 0 0 0

((c&p.+d&p.)*(c&p.-d&p.)) x
_648 _48 0 0 0

f=: c&p. (+*-) d&p.
f x
_648 _48 0 0 0

lg=: pD c pp d
5 20 30 20 5

(g&p. ,: (c&p.*d&p.) D) x
405 80 5 0 5
405 80 5 0 5

```

## 25. Polynomials in Terms of Roots

The product  $\ast/y-r$  is called a *polynomial in terms of the roots*  $r$ , because it can also be expressed as a polynomial applied to the argument  $y$ , and because  $r$  is the list of *roots* or *zeros* of the resulting function. For example:

```

    */y-r [ y=: 7 [ r=: 2 3 5 [ x=: 7 6 5 4 3 2
40
    pp=: +//. @ (*/)
    c=: pp/monomials=: (- ,. 1:) r
    cfr=: [: pp/ - ,. 1:      Coefficients from roots
    pir=: */@([ ]-[ ])"1 0    Polynomial in terms of roots
    ,.&.>(r;monomials;c;(cfr r);(c&p. y);(r pir x))
+---+---+---+---+---+---+
| 2 | _2 1 | _30 | _30 | 40 | 40 |
| 3 | _3 1 | 31 | 31 |   | 12 |
| 5 | _5 1 | _10 | _10 |   | 0  |
|   |   | 1 | 1 |   | _2 |
|   |   |   |   |   | 0  |
|   |   |   |   |   | 0  |
+---+---+---+---+---+---+

```

Since the last (highest order) coefficient produced by `cfr` is necessarily 1, the function `pir` cannot produce a general polynomial, but it can if provided with a multiplier. We therefore re-define `cfr` and `pir` to apply to a boxed list of multiplier and roots as follows:

```

    CFR=: (* cfr)&>/
    PIR=: CFR@[ p. ]
    CFR 3;r
_90 93 _30 3
    (3;r) PIR x
120 36 0 _6 0 0

```

We now illustrate the use of a polynomial in approximation:

```

    ]ce=: ^ t. i. 7      First seven terms of Taylor series for exponential
1 1 0.5 0.166667 0.0416667 0.00833333 0.00138889
    (^ - ce&p.) _1 _0.5 0 0.5 1      Comparison with exponential
_0.000176114 _1.45834e_6 0 1.65264e_6 0.000226273
    pD ce      The exponential function equals its own derivative
1 1 0.5 0.166667 0.0416667 0.00833333

```

## 26. Polynomials: Roots from Coefficients (Newton's Method)

Because the polynomials  $(m;r)\&PIR$  and  $(c=:CFR(m;r))\&p.$  are identical, the parameters  $m;r$  and  $c$  are said to be different *representations* of the same function. Each representation has its own useful properties. For example, addition of polynomials is easy in the coefficient representation but difficult in the root representation; the identification of the zeros of the function is difficult in the coefficient representation but trivial in the root representation. It is therefore useful to have functions that transform each representation to the other.  $CFR$  serves for one direction; the inverse problem is approached by methods of successive approximation.

For any function  $f$ , the difference  $(f\ r)-(f\ a)$  for nearby points  $r$  and  $a$  is approximately equal to the difference  $r-a$  multiplied by the slope of the tangent to the graph of  $f$  at the point  $a, f\ a$ , that is, the derivative of  $f$  at  $a$ . Conversely, the difference

$r-a$  is approximated by  $((f\ r)-(f\ a))\%f\ D\ a$ , and  $r$  is approximated by  $a+((f\ r)-(f\ a))\%f\ D\ a$ .

If  $f$  is the polynomial  $c\&p.$  and  $r$  is one of its roots, then  $f\ r$  is zero, and if  $a$  is an approximation to  $r$ , the expression for  $r$  reduces to  $a-(f\ a)\%f\ D\ a$ . This may provide a better approximation to  $r$ , and is embodied in *Newton's method*, defined as an adverb, and illustrated as follows:

```
newton=: 1 : 0
] - x. % x.D
)

f=: (c=: 12 _10 2)\&p.

f a=: 2.4    f newton a
_0.48 1.2
f newton ^:0 1 2 3 4 _ a    f 2
2.4 1.2 1.75385 1.9594 1.99848 2 0
]a=: (^ - 4:) newton ^: 0 1 2 3 _ a=: 1
1 1.47152 1.38982 1.3863 1.38629
^ {: a
4
```

For the particular case of polynomials, we may define an adverb that applies to coefficients and uses the polynomial derivative  $pD$  instead of the general derivative  $D$ :

```
pD=: 1: }. ] * i.@#
NEWTON=: 1 : ' ] - x.&p. % (pD x.)&p. '

c NEWTON ^:0 1 2 3 4 _ a=: 2.4
2.4 1.2 1.75385 1.9594 1.99848 2
```

## 27. Polynomials: Roots from Coefficients (Kerner's Method)

Newton's method applies to one root at a time and requires a good starting approximation. Kerner's method is a generalization that gives all the roots, starting from a *list* `a` and dividing each element of the residual `f a` by the derivative with respect to the corresponding root. It applies only to a polynomial whose highest order coefficient is 1, and we first normalize the coefficients by dividing by the last, yielding a polynomial having the same roots. The method converges to complex roots only if at least one of the initial approximations is complex. We will use the Taylor series approximation to the exponential function, because the corresponding polynomial has complex roots:

```
d=: ^ t. i.6
1 1 0.5 0.166667 0.0416667 0.00833333

]c=: (norm=: % {:) d
120 120 60 20 5 1

+. a=: (init=: r.@}.@i.@#) c          |a
0.540302 0.841471                      1 1 1 1 1
_0.416147 0.909297
_0.989992 0.14112
_0.653644 _0.756802
0.283662 _0.958924

deriv=: [: */ 0&=@{.}%(-/~ ,: 1:)
kerner=: 1 : 0
] - x.&p. % deriv@]
)

r=: c kerner ^:_ a
+.(/:|) r                                Real and imag parts of roots sorted by magnitude
_2.18061 4.57601e_31
_1.6495 1.69393
_1.6495 _1.69393
0.239806 3.12834
0.239806 _3.12834

>./|c p. r
1.04488e_13
```

These results may be compared with the use of the primitive `p.` on the un-normalized coefficients `d`. Thus:

```
p. 2 4 2
+-+-----+
|2|_1 _1|
+-+-----+
```



```

    ,.:}.p. d
0.2398064j3.12834
0.2398064j_3.12834
_1.6495j1.69393
_1.6495j_1.69393
_2.18061

```

Newton's method may also be used for a complex root:

```

+. d NEWTON ^:0 1 2 3 _ a=: 1j1
      1      1
0.0166065 0.99639
_0.990523 0.992532
_1.95338 1.10685
_1.6495 1.6939

```

## 28. Polynomials: Stopes

The expression  $x + s * i. n$  is often called a *factorial function* but, to avoid confusion with the function  $!$ , we will call it a *stope*; the factors occurring in its definition differ by steps of size  $s$ , like the steps in a mine stope. Stopes are useful in actuarial work and in the difference calculus.

The stope is a variant of the power  $x^n$  (being equivalent for the case  $s=:0$ ), and is provided by the fit conjunction in the variant  $^!.s$ . For example:

```

x + s * i. n [ x=: 7 [ n=: 5 [ s=: _1
7 6 5 4 3

(* / x + s * i. n); (x ^!.s n); (x ^!.0 n); (x^n)
+-----+-----+-----+-----+
| 2520 | 2520 | 16807 | 16807 |
+-----+-----+-----+-----+

```

The phrase  $+ / c * x ^!.s i. \# c$  is called a *stope polynomial*, just as  $+ / c * x ^i. \# c$  may be called a *power polynomial*. We define an adverb  $P$  that applies to any step size  $s$  to provide the corresponding stope polynomial:

```

P=: 1 : '+/@([ * ] ^!. x. i. @#@[) "1 0'
c=: 1 3 3 1 [ d=: 1 7 6 1 [ x=: 0 1 2 3 4
(c p. x); (c 0 P x); (d _1 P x); (d p. !_1 x)
+-----+-----+-----+-----+
| 1 8 27 64 125 | 1 8 27 64 125 | 1 8 27 64 125 | 1 8 27 64 125 |
+-----+-----+-----+-----+

```

As illustrated above, stoep polynomials are (for a suitable choice of coefficients) equivalent to ordinary polynomials. Moreover, the transformations between them are provided by a matrix product as follows:

```

VM=: 1 : '[ ^!.x./ i.@#@]'
TO=: 2 : '(x. VM %. y. VM)~ @i.@#'
(0 TO _1 c) +/ . * c
1 7 6 1

```

The matrices  $0 \text{ TO } _1$  and  $_1 \text{ TO } 0$  are mutually inverse, and are simply related to *Stirling* numbers:

```

(0 TO _1 i.5);(_1 TO 0 i.5)
+-----+-----+
| 1 0 0 0 0 | 1 0 0 0 0 |
| 0 1 1 1 1 | 0 1 _1 2 _6 |
| 0 0 1 3 7 | 0 0 1 _3 11 |
| 0 0 0 1 6 | 0 0 0 1 _6 |
| 0 0 0 0 1 | 0 0 0 0 1 |
+-----+-----+

```

The stoep polynomial is also provided by the variant `p.!.s.`

# Dictionary

**J** is a dialect of APL, a formal imperative language. Because it is imperative, a sentence in **J** may also be called an *instruction*, and may be *executed* to produce a *result*. Because it is formal and unambiguous it can be executed mechanically by a computer, and is therefore called a *programming language*. Because it shares the analytic properties of mathematical notation, it is also called an analytic language.

APL originated in an attempt to provide consistent notation for the teaching and analysis of topics related to the application of computers, and developed through its use in a variety of topics, and its implementation in computer systems [1-5].

**J** is implemented in C (as detailed in Hui [6]), and is ported to a number of different host computer systems. The effect of the specific host is minimal, and communication with it is confined to the single *foreign conjunction* detailed in the Appendix. See help files for other host facilities such as Windows.

The Introduction in this book provides guidance to beginners. References [7-9] use **J** in the exposition of various mathematical topics.

## ***I: Alphabet and Words***

The alphabet is standard ASCII, comprising *digits*, *letters* (of the English alphabet), the *underline* (used in names and numbers), the (single) *quote*, and others (which include the space) to be referred to as *graphics*. Alternative spellings for the *national use* characters (which differ from country to country) are discussed under *Alphabet(a)*.

Numbers are denoted by digits, the underbar (for negative signs and for infinity and minus infinity — when used alone or in pairs), the period (used for decimal points and *necessarily* preceded by one or more digits), the letter e (as in 2.4e3 to signify 2400 in exponential form), and the letter j to separate the real and imaginary parts of a complex number, as in 3e4j\_0.56. Also see the discussion of *Constants*.

A numeric *list* or *vector* is denoted by a list of numbers separated by spaces. A list of ASCII characters is denoted by the list enclosed in single quotes, a pair of adjacent single quotes signifying the quote itself: 'can' 't' is the five-character abbreviation of the six-character word 'cannot'. The *ace* a: denotes the boxed empty list <\$0.

*Names* (used for pronouns and other surrogates, and assigned referents by the copula, as in prices=: 4.5 12) begin with a letter and may continue with letters, underlines, and digits. A name that ends with an underline or that contains two consecutive under-lines is a *locative*, as discussed in Section II.I.



A *primitive* or *primary* may be denoted by a single graphic (such as `+` for *plus*) or by a graphic modified by one or more following *inflections* (a period or colon), as in `+`, `+`, and `+` for *or* and *nor*. A primary may also be an inflected name, as in `e.` and `o.` for *membership* and *pi times*. A primary cannot be assigned a referent.

## II. Grammar

The following sentences illustrate the six parts of speech:

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

```
prices=: 3 1 4 2
orders=: 2 0 2 1
orders * prices
6 0 8 2
```

### PARTS of SPEECH

<code>+/orders*prices</code>	<code>50 fahrenheit</code>	Nouns/Pronouns
<code>16</code>		
<code>+/ \ 1 2 3 4 5</code>	<code>+ - * % bump</code>	Verbs/Proverbs
<code>1 3 6 10 15</code>	<code>/ \</code>	Adverbs
<code>bump=: 1&amp;+</code>	<code>&amp;</code>	Conjunction
<code>bump prices</code>	<code>( )</code>	Punctuation
<code>4 2 5 3</code>	<code>=:</code>	Copula

Verbs act upon nouns to produce noun results; the nouns to which a particular verb applies are called its *arguments*. A verb may have two distinct (but usually related) meanings according to whether it is applied to one argument (to its right), or to two arguments (left and right). For example, `2%5` yields `0.4`, and `%5` yields `0.2`.

An adverb acts on a single noun or verb to its *left*. Thus, `+/` is a derived verb (which might be called *plus over*) that sums an argument list to which it is applied, and `* /` gives the product of a list. A conjunction applies to two arguments, either nouns or verbs.

Punctuation is provided by parentheses that specify the sequence of execution as in elementary algebra; other punctuation includes `if.` `do.` `end.` as discussed under Explicit Definition (`:`).

The word `=:` behaves like the copulas “is” and “are” in English, and is read as such, as in “area is 3 times 4” for `area=: 3*4`. The name `area` thus assigned is a *pronoun* and, as in English, it plays the role of a noun. Similar remarks apply to names assigned to verbs,

adverbs, and conjunctions. Entry of a name alone displays its value. Errors are discussed in Section II.J (Errors and Suspension).

## A. 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 math, *scalar*, *vector*, and *matrix*. Numbers and literals are represented as stated in Part I.

**Arrays.** A single entity such as 2.3 or \_2.3j5 or 'A' or '+' is called an atom. The verb denoted by comma chains its arguments to form a list whose *shape* (given by the verb \$) is equal to the number of atoms combined. For example:

```
$ date=: 1,7,7,6
4
word=: 's','a','w'
|. word          |. date
was              6 7 7 1
```

The verb |. used above is called *reverse*. The phrase s\$b produces an array of shape s from the list b. For example:

```
(3,4) $ date,1,8,6,7,1,9,1,7
1 7 7 6
1 8 6 7
1 9 1 7
table=: 2 3$ word,'bat'
table          $table
saw            2 3
bat
```

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. For example, axis 0 of table has length 2 and axis 1 has length 3.

The last *k* axes of an array b determine *rank-k cells* or *k-cells* of b. The rest of the shape vector is called the frame of b relative to the cells of rank *k*; if \$c is 2 3 4 5, then c has the frame 2 3 relative to cells of rank 2, the frame 2 3 4 5 relative to 0-cells (atoms), and an empty frame relative to 4-cells. If:

```
] b=: 2 3 4 $ 'abcdefghijklmnopqrstuvw'
abcd
efgh
ijkl
```

```
mnop
qrst
uvwx
```

then the list `abcd` is a 1-cell of `b`, and the letters are each 0-cells.

A cell of rank one less than the rank of `b` is called an *item* of `b`; an atom has one item, itself. For example, the verb *from* (denoted by `{`) selects items from its right argument, as in:

```

      0{b          1{b          0{0{b
abcd             mnop         abcd
efgh
ijkl

      2 1{0{b      1{2{0{b      0{3
ijkl              j             3
efgh
```

Moreover, the verb *grade* (denoted by `/:`) provides indices to `{` that bring items to “lexical” order. Thus:

```

      g=: /: n=: 4 3$3 1 4 2 7 9 3 2 0
      n              g              g{n
3 1 4              1 0 3 2          2 7 9
2 7 9              3 1 4
3 2 0              3 1 4
3 1 4              3 2 0
```

Negative numbers, as in `_2`-cell and `_1`-cell (an item), are also used to refer to cells whose *frames* are of the length indicated by the magnitude of the number. For example, the list `abcd` may be referred to either as a `_2`-cell or as a 1-cell of `b`.

**Open and Boxed.** The nouns discussed thus far are called *open*, to distinguish them from *boxed* nouns produced by the verb *box* denoted by `<`. 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. Thus:

```

      words=: (<'I'), (<'was'), (<'it')
      letters=: 'I was it'
      $words          $letters
3                      8

      |. words          |. letters
+---+---+---+          ti saw I
|it|was|I|
+---+---+---+
      2 3$words, |. words
+---+---+---+
```

104

```
| I | was | it |  
+--+-----+--+  
| it | was | I |  
+--+-----+--+
```



## B. Verbs

**Monads and Dyads.** Verbs have two definitions, one for the *monadic* case (one argument), and one for the *dyadic* case (two arguments). The dyadic definition applies if the verb is preceded by a suitable left argument, that is, any noun that is not itself an argument of a conjunction; otherwise the monadic definition applies.

The monadic case of a verb is also called a *monad*, and we speak of the *monad* % used in the phrase %4, and of the *dyad* % used in 3%4. Either or both may have empty domains.

**Ranks of Verbs.** The notion of verb rank is closely related to that of noun rank: a verb of rank  $k$  applies to each of the  $k$ -cells of its argument. For example (using the array `b` from Section A):

```
,b
abcdefghijklmnopqrstuvwxyz

,"2 b                                ,"_1 b
abcdefghijklmnopqrstuvwxyz      abcdefghijkl
mnopqrstuvwxyz                    mnopqrstuvwxyz
```

Since the verb *ravel* (denoted by `,`) applies to its entire argument, its rank is said to be *unbounded*. The *rank* conjunction `"` used in the phrase `, "2` produces a related verb of rank 2 that ravel each of the 2-cells to produce a result of shape 2 by 12.

The shape of a result is the frame (relative to the cells to which the verb applies) catenated with the shape produced by applying the verb to the individual cells. Commonly these individual shapes agree, but if not, they are first brought to a common rank by introducing leading unit axes to any of lower rank, and are then brought to a common shape by *padding* with an appropriate *fill* element: space for a character array, 0 for a numeric array, and a boxed empty list for a boxed array. For example:

```
i."0 s=: 2 3 4                                >'I'; 'was'; 'here'
0 1 0 0                                          I
0 1 2 0                                          was
0 1 2 3                                          here
```

The dyadic case of a verb has two ranks, governing the left and right arguments. For example:

```
p=: 'abc'
q=: 3 5$'wake read lamp '
p,"0 1 q
awake
```

106

bread  
clamp

Finally, each verb has three intrinsic ranks: monadic, left, and right. The definition of any verb need specify only its behaviour on cells of the intrinsic ranks, and the extension to arguments of higher rank occurs systematically. The ranks of a verb merely place upper limits on the ranks of the cells to which it applies; its domain may include arguments of lower rank. For example, matrix inverse ( $\%.$ ) has monadic rank 2, but treats the case of a vector as a one-column matrix.

**Agreement.** In the phrase  $p \ v \ q$ , the arguments of  $v$  must *agree* in the sense that one frame must be a prefix of the other, as in  $p, "0 \ 1 \ q$  above, and in the following examples:

	$p, " \ 1 \ 1 \ q \ 3 \ 4 \ 5 * i. \ 3 \ 4$
abcwake	0 3 6 9
abcread	16 20 24 28
abclamp	40 45 50 55

$(i. \ 3 \ 4) * 3 \ 4 \ 5$
0 3 6 9
16 20 24 28
40 45 50 55

If a frame contains 0, the verb is applied to a cell of fills. For example:

```
( $ # " 2 i. 1 0 3 4 ); ( $ 2 3 % " 1 i. 0 2 )
+---+---+
| 1 0 | 0 2 |
+---+---+

( $ $ " 2 i. 1 0 3 4 ); ( $ 2 3 % / " 1 i. 0 4 )
+-----+-----+
| 1 0 2 | 0 2 4 |
+-----+-----+
```

## C. Adverbs and Conjunctions

Unlike verbs, adverbs and conjunctions have fixed valence: an adverb is monadic (applying to a single argument to its *left*), and a conjunction is dyadic.

Conjunctions and adverbs apply to noun or verb arguments; a conjunction may produce as many as four distinct classes of results.

For example,  $u \& v$  produces a *composition* of the verbs  $u$  and  $v$ ; and  $\wedge 2$  produces the *square* by bonding the power function with the right argument 2; and  $2 \& \wedge$  produces the function *2-to-the-power*. The conjunction  $\&$  may therefore be referred to by different names for the different cases, or it may be referred to by the single term *and* (or *with*), which roughly covers all cases.

## D. Comparatives

The comparison  $x=y$  is treated like the everyday use of equality (that is, with a reasonable relative tolerance), yielding 1 if the difference  $x-y$  falls relatively close to zero. Tolerant comparison also applies to other relations and to *floor* and *ceiling* ( $<.$  and  $>.$ ); a precise definition is given in Part III under *equal* ( $=$ ). An arbitrary tolerance  $\epsilon$  can be specified by using the *fit* conjunction ( $!$ ), as in  $x = !.\epsilon y$ . The global tolerance can be queried and set using  $\$! : 18$  and  $\$! : 19$ .

## E. Parsing and Execution

A sentence is evaluated by executing its phrases in a sequence determined by the parsing rules of the language. For example, in the sentence  $10\%3+2$ , the phrase  $3+2$  is evaluated first to obtain a result that is then used to divide  $10$ . In summary:

1. Execution proceeds from right to left, except that when a right parenthesis is encountered, the segment enclosed by it and its matching left parenthesis is executed, and its result replaces the entire segment and its enclosing parentheses.
2. Adverbs and conjunctions are executed before verbs; the phrase  $, "2-a$  is equivalent to  $( , "2) -a$ , not to  $, "(2-a)$ . Moreover, the left argument of an adverb or conjunction is the entire verb phrase that precedes it. Thus, in the phrase  $+ / . * / b$ , the rightmost adverb  $/$  applies to the verb derived from the phrase  $+ / . *$ , not to the verb  $*$ .
3. A verb is applied dyadically if possible; that is, if preceded by a noun that is not itself the right argument of a conjunction.
4. Certain *trains* form verbs, adverbs, and conjunctions, as described in § F.
5. To ensure that these summary parsing rules agree with the precise parsing rules prescribed below, it may be necessary to parenthesize an adverbial or conjunctival phrase that produces anything other than a noun or verb.

One important consequence of these rules is that in an unparenthesized expression the right argument of any verb is the result of the entire phrase to its right. The sentence  $3 * p \% q ^ | r - 5$  can therefore be *read* from left to right: the overall result is 3 times the result of the remaining phrase, which is the quotient of  $p$  and the part following the  $\%$ , and so on.

Parsing proceeds by moving successive elements (or their *values* except in the case of proverbs and names immediately to the left of a copula) from the tail end of a *queue* (initially the original sentence prefixed by a left marker **S**) to the top of a *stack*, and eventually executing some eligible portion of the stack and replacing it by the result of the execution. For example, if  $a = 1\ 2\ 3$ , then  $b = +/2*a$  would be parsed and executed as follows:

<b>S</b>	$b = + / 2 * a$			
<b>S</b>	$b = + / 2 *$		1	2
<b>S</b>	$b = + / 2$		*	1
<b>S</b>	$b = + /$		2	* 1
<b>S</b>	$b = +$		/	2 * 1
<b>S</b>	$b =$			2 4
<b>S</b>	$b$			6
<b>S</b>				12
<b>S</b>				12
<b>S</b>				12
<b>S</b>				12

The foregoing illustrates two points: 1) Execution of the phrase  $2 * 1\ 2\ 3$  is deferred until the next element (the  $/$ ) is transferred; had it been a conjunction, the  $2$  would have been *its* argument, and the monad  $*$  would have applied to  $1\ 2\ 3$ ; and 2) Whereas the *value* of the name  $a$  moves to the stack, the name  $b$  (because it precedes a copula) moves unchanged, and the pronoun  $b$  is assigned the value 12.

Parsing can be observed using the trace conjunction  $13! : 16$  (q.v.).

The executions in the stack are confined to *the first four elements* only, and eligibility for execution is determined only by the *class* of each element (noun, verb, etc., an unassigned name being treated as a verb), as prescribed in the following parse table. The classes of the first four elements of the stack are compared with the first four columns of the table, and the first row that agrees in all four columns is selected. The bold italic elements in the row are then subjected to the action shown in the final column, and are replaced by its result. If no row is satisfied, the next element is transferred from the queue.

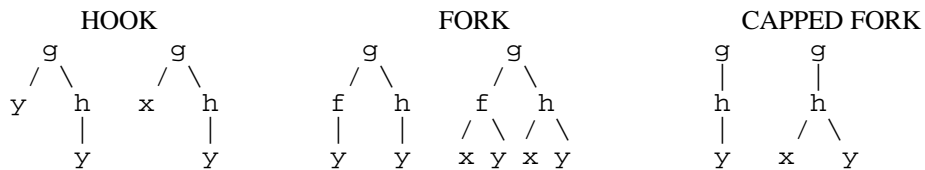
EDGE	<b>VERB</b>	<b>NOUN</b>	ANY	0 Monad
EDGE+AVN	VERB	<b>VERB</b>	<b>NOUN</b>	1 Monad
EDGE+AVN	<b>NOUN</b>	<b>VERB</b>	<b>NOUN</b>	2 Dyad
EDGE+AVN	<b>VERB+NOUN</b>	<b>ADV</b>	ANY	3 Adverb
EDGE+AVN	<b>VERB+NOUN</b>	<b>CONJ</b>	<b>VERB+NOUN</b>	4 Conj
EDGE+AVN	<b>VERB</b>	<b>VERB</b>	<b>VERB</b>	5 Trident
EDGE	<b>CAVN</b>	<b>CAVN</b>	<b>CAVN</b>	6 Trident
EDGE	<b>CAVN</b>	<b>CAVN</b>	ANY	7 Bident
<b>NAME+NOUN</b>	<b>ASGN</b>	<b>CAVN</b>	ANY	8 Is
<b>LPAR</b>	<b>CAVN</b>	<b>RPAR</b>	ANY	9 Paren
Legend:	AVN CAVN EDGE	denotes denotes denotes	ADV+VERB+NOUN CONJ+ADV+VERB+NOUN MARK+ASGN+LPAR	

## F. Trains

An isolated sequence, such as  $(+ \ * \ /)$ , which the “normal” parsing rules (other than the three labelled *trident* and *bident*) do not resolve to a single part of speech is called a *train*, and may be further resolved as described below.

Meanings are assigned to certain trains of two or three elements and, by implication, to trains of any length by repeated resolution. For example, the trains  $+ \ - \ * \ \%$  and  $+ \ - \ * \ \% \ ^$  are equivalent to  $+ \ ( \ - \ * \ \% )$  and  $+ \ - \ ( \ * \ \% \ ^ )$ .

A *verb* is produced by trains of three or two verbs, as defined by the following diagrams:



For example,  $5(+ \ * \ -)3$  is  $(5+3) \ * \ (5-3)$ . If *f* is a cap ( $[ : )$ ) the capped branch simplifies the forks to  $g \ h \ y$  and  $g \ x \ h \ y$ .

The ranks of the hook and fork are infinite.

Trains may also produce adverbs and conjunctions, and trains of two and three elements are called *bidents*, and *tridents*, respectively; hooks and forks are special cases. Tree displays illustrate the choice of the names fork and trident:

```

mean=: +/ % #
conj=: ]. , [.
tree=: 5!:4@<

tree 'mean'
+- / --- +
--+- %
+- #

tree 'conj'
+- ].
--+- ,
+- [.
```

The following tables define all possible tridents and bidents, using light italics to denote the optional left arguments of (ambivalent) verbs :

N0 V1 N2	noun	<i>x</i> V1 <i>y</i>
V0 V1 V2	verb	( <i>x</i> V0 <i>y</i> ) V1 ( <i>x</i> V2 <i>y</i> )
V0 V1 C2	conj	V0 V1 ( <i>x</i> C2 <i>y</i> )
A0 V1 V2	adv	( <i>x</i> A0) V1 V2
C0 V1 V2	conj	( <i>x</i> C0 <i>y</i> ) V1 V2
C0 V1 C2	conj	( <i>x</i> C0 <i>y</i> ) V1 ( <i>x</i> C2 <i>y</i> )
A0 A1 V2	conj	( <i>x</i> A0) ( <i>y</i> A1) V2
A0 A1 A2	adv	(( <i>x</i> A0) A1) A2
C0 A1 A2	conj	(( <i>x</i> C0 <i>y</i> ) A1) A2
N0 C1 N2	verb	<i>x</i> (N0 C1 N2) <i>y</i>
N0 C1 V2	verb	<i>x</i> (N0 C1 V2) <i>y</i>
N0 C1 A2	adv	N0 C1 ( <i>x</i> A2)
N0 C1 C2	conj	N0 C1 ( <i>x</i> C2 <i>y</i> )
V0 C1 N2	verb	<i>x</i> (V0 C1 N2) <i>y</i>
V0 C1 V2	verb	<i>x</i> (V0 C1 V2) <i>y</i>
V0 C1 A2	adv	V0 C1 ( <i>x</i> A2)

V0 C1 C2	conj	V0 C1 (x C2 y)
A0 C1 N2	adv	(x A0) C1 N2
A0 C1 V2	adv	(x A0) C1 V2
A0 C1 A2	conj	(x A0) C1 (y A2)
A0 C1 C2	conj	(x A0) C1 (x C2 y)
C0 C1 N2	conj	(x C0 y) C1 N2
C0 C1 V2	conj	(x C0 y) C1 V2
C0 C1 A2	conj	(x C0 y) C1 (y A2)
C0 C1 C2	conj	(x C0 y) C1 (x C2 y)
N0 A1	verb	x (N0 A1) y
N0 C1	adv	N0 C1 x
V0 N1	noun	V0 y
V0 V1	verb	x (or y) V0 V1 y
V0 A1	verb	x (V0 A1) y
V0 C1	adv	V0 C1 x
A0 V1	adv	(x A0) V1
A0 A1	adv	(x A0) A1
A0 C1	adv	(x A0) C1 x
C0 N1	adv	x C0 N1
C0 V1	adv	x C0 V1
C0 A1	conj	(x C0 y) A1

## G. Extended and Rational Arithmetic

Extended precision integer constants can be entered as a sequence of decimal digits terminated by an `x`. The monad `x:` applies to integers and produces extended integers. For example, the 2-element vector `1234x 56x` (or `1234 56x`) is equivalent to `x: 1234 56`. Various primitives produce extended results if the argument(s) are extended. Some functions `f` produce floating point (inexact) results on some extended arguments because the result is not integral; however, `<.@f` and `>.@f` produce extended integer results when applied to extended integer arguments. Comparisons involving extended integers are exact. For example:



Rational constants can be entered as the decimal digits of the numerator and denominator, separated by an `r` and preceded by an optional sign. Thus `3r4` is the rational number three-quarters and `_12r5` is negative 12 divided by 5. Rational numbers are stored and displayed in a standard form, with the numerator and denominator relatively prime and the denominator positive. Thus:

Various primitive functions produce (exact) rational results if the argument(s) are rational; non-rational functions produce (inexact) floating point or complex results when applied to rationals, if the function only has a limited number of rational arguments that produce rational results. (For example, `%:y` is rational if the atoms of `y` are perfect squares; `^0x1` is floating point.) The quotient of two extended integers is an extended integer (if evenly divisible) or rational (if not). Comparisons involving two rationals are non-tolerant (exact). Functions or operators that require integer arguments (such as the left arguments of `{ { . # $ }`) also accept rational arguments, if they are integers. Other dyadic functions (e.g. `+ - * % , = <`) convert their arguments to the same type, according to the following table:

	B	I	X	Q	D	Z	
B	B	I	X	Q	D	Z	B - boolean
I	I	I	X	Q	D	Z	I - integer
X	X	X	X	Q	D	Z	X - extended integer
Q	Q	Q	Q	Q	D	Z	Q - rational
D	D	D	D	D	D	Z	D - floating point
Z	Z	Z	Z	Z	Z	Z	Z - complex

For example, in the expression `2.5+1r2`, the `1r2` is converted to `0.5` before being added to `2.5`, resulting in a floating point `3`. And in the expression `2+1r2`, the `2` is converted to `2r1` before being added to `1r2`, resulting in `5r2`.

In particular, a comparison involving a rational and a floating point number is tolerant, because the rational argument is first converted into a floating point number.

The verb `x:` (q.v.) produces rational approximations to non-rational arguments.

```
2%3
0.666667
```

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

```
(+%) /\ 10$1                                Floating point convergents to golden mean
1 2 1.5 1.66667 1.6 1.625 1.61538 1.61905 1.61765 1.61818
```

```
(+%) /\ x: 10$1                                Rational versions of same
1 2 3r2 5r3 8r5 13r8 21r13 34r21 55r34 89r55
```

```
|: 2 x: (+%) /\ x: 10$1
1 2 3 5 8 13 21 34 55 89
1 1 2 3 5 8 13 21 34 55
```

```
(+%) / 100$1r1
573147844013817084101r354224848179261915075
```

```
0j30 " : (+%) / 100$1r1                        Display 30 decimal places
1.618033988749894848204586834366
```

```
H=: % @: >: @: (+/~) @: i. @ x:                Hilbert matrix of order n
] h=: H 6
1 1r2 1r3 1r4 1r5 1r6
1r2 1r3 1r4 1r5 1r6 1r7
1r3 1r4 1r5 1r6 1r7 1r8
1r4 1r5 1r6 1r7 1r8 1r9
1r5 1r6 1r7 1r8 1r9 1r10
1r6 1r7 1r8 1r9 1r10 1r11
```

<code>-/ .* h</code> <code>1r186313420339200000</code>	Determinant of $h$
<code>~. q: % -/ .* h</code> <code>2 3 5 7 11</code>	Unique prime factors of reciprocal of $\det$
<code>i.&amp;.(p:^:_1) 2*6</code> <code>2 3 5 7 11</code>	Primes less than $2*n$
<code>^ 2r1</code> <code>7.38906</code>	$^y$ is floating point or complex
<code>%: 49r25</code> <code>7r5</code>	<code>%:</code> on a rational perfect square is rational
<code>%: 49r25 10r9</code> <code>1.4 1.05409</code>	
<code>%: _2r1</code> <code>0j1.41421</code>	
<code>1=1+10r1^_15</code> <code>0</code>	Exact (rational) comparison
<code>(1.5-0.5) = 1+10r1^_15</code> <code>1</code>	Tolerant (floating point) comparison
<code>0.5 = 1r2</code> <code>1</code>	

## H. Frets and Scripts

Host systems commonly use the “line-feed” or “carriage return” characters `10{a.` or `13{a.` (or both together) as *frets* to mark divisions into individual *lines*. A character list provided with zero or more frets will be called a *script*.

As detailed in the Appendix, a script `t` may be filed and retrieved by expressions of the form `t 1!:2 <'abc'` and `t=: 1!:1 <'abc'`, and may be *executed* by the expression `0!:11 <'abc'`.

Convenient entry of scripts is provided by the phrase `0 : 0`; succeeding keystrokes are accepted as literal characters; the *enter* key that would normally terminate the entry is accepted as a fret; and the entry is terminated by a lone right parenthesis that is accepted as a final fret. For example:

116

```

s=: 0 : 0
y.*%:y.
:
x.*!y.
)
a. i. s
121 46 42 37 58 121 46 10 58 10 120 46 42 33 121 46 10

```

The character with index 10 marks the end of each line

Boxed and table representations of a script `s` may be obtained as follows:

```

]b=: </:_2 s
+-----+-----+
|y.*%:y. | : |x.*!y. |
+-----+-----+
]t=: >b
y.*%:y.
:
x.*!y.

```

Cut on the final fret and exclude the frets

Any one of these representations `r` may be used as the right argument to the explicit definition conjunction to produce an adverb (`1 : r`), conjunction (`2 : r`), or verb (`3 : r` or `4 : r`). For example:

```

f=: 3 : s
f 9
27
3 f 4
72

```

The colon in the script separates the monadic and dyadic cases

The `x.` and `y.` refer to the left and right arguments

The phrases `a=: 1 : 0` and `c=: 2 : 0` and `v=: 3 : 0` provide direct entry of adverbs, conjunctions, and verbs.

Files of scripts may define functions and other entities that can serve to supplement the primaries of **J**. They are commonly referred to as *secondary* or *tertiary* functions according to their relative generality.

## I. Locatives

Locative `abc_f_` refers to `abc` in locale `f`; indirect locative `abc__xy` refers to `abc` in the locale whose name is the current value of `xy`. For compatibility with previous versions, the non-standard `abc__` is accepted and is the same as `abc_base_`. Thus:

```

b=: 1
Rome=: 2
Rome_NewYork_=: 20
f_NewYork_=: 3 : '3*b=: Rome+y.'
f_NewYork_ 10
90

b,Rome
1 2

b_NewYork_
30

```

A name is global if it is not assigned by `=:` within Explicit Definition (`:`). Every global name is executed in the current locale. Initially, the current locale is `base`. A locale `f_abc_`, while it is executing, switches the current locale to `abc`. The verb `18!:4` also switches the current locale, and `18!:5` gives its name.

The name `f_abc_` is *executed in* locale `abc` in the sense that a global name referenced in `f` is sought therein and, if not found, is then sought in the locales in the path of `abc` (but is still executed in `abc`). The path of a locale is initially `,<,'z'`, except that locale `z` has an empty path initially, and may be changed using `18!:2`.

A locale is commonly populated by a script, by appropriate naming of the verb used to execute the script. For example, if the file `stats` contains the script:

```

mean=: sum % #
sum=: +/

```

Then:

```

ssx_z_=: 0!:10      Silent script execution

ssx_a_ <'stats'      Populate locale a

mean=: 'in base locale'

mean_a_ 3 4 5
4

ssx_bc_ <'stats'      Populate locale bc
sum_bc_ 3 4 5
12

```

The example also illustrates the use of locale paths, in this case the `z` locale: First, the utility `ssx` is defined in the `z` locale. In executing `ssx_a_`, `ssx` is not found in locale `a` and is therefore sought (and found) in locale `z`. Since `ssx_a_` is *executed* in locale `a`, the names in the `stats` script are *defined* in locale `a`, populating it thereby. Similarly for `ssx_bc_`.

See also 18! : in the Appendix and the “Locales” and “Object Oriented Programming” labs distributed with the system.

## J. Errors and Suspension

Execution of a sentence *suspends* when an error occurs, and an error message and context information are then displayed. Four blanks indicate where parsing stopped. Suspension may occur in immediate execution, in the execution of a script file, or in the execution of a user-defined verb, adverb, or conjunction, as illustrated by the following examples:

### Immediate execution

```
2+'a'
|domain error
| 2    +'a'
```

### Execution of a script file

```
t=: '2*3',(10{a.),'2+'a''',(10{a.),'2+3'
t
2*3
2+'a'
2+3
t 1!:2 <'test'
0!:011 <'test'
2*3
6
2+'a'
2+3
5
0!:001 <'test'
2*3
6
2+'a'
|domain error
| 2    +'a'
|[-2]
```

A script

Write script file

Execute file, continue on error, display (011)

Execute file, stop on error, display (001)

### User-Defined Verb

```
g=: 3 : ('1+y.' ; ':' ; '2+x.+y.')
3+g 'a'
|domain error: g
| 1    +y.
13! :0 (1)
```

Enable suspension

```

      3+g 'a'
|domain error: g
|  1      +y.
|g[0]

```

<pre>       y. a       y.=. 12       13!:4 '' 16 </pre>	<pre> six-space indent indicates suspension  Redefine local value of y. Resume execution at the current line Result using redefined value of y. </pre>
---	--

Sentences can be executed in the suspended environment, local values can be accessed, and execution can be resumed. Errors cause suspension *only* if suspension is enabled (by the phrase `13!:0`). When suspension is in effect, the input prompt is six spaces.

Suspension and debugging facilities are controlled by the `13! :` family as described in the Appendix.

### III. Definitions

Each main entry in the body of the dictionary is headed by a line beginning with the informal name of the monadic case of the function, and ending with the informal name of the dyadic case. The line also contains the formal name of the function, consisting of one character or two, the last of which is a period or colon. In the case of a conjunction, the formal name is preceded by *m* or by *u* (denoting a noun or a verb argument) and is followed by *n* or *v*. An adverb has no symbol to its right.

The three ranks (in the order *monadic*, *left*, and *right*) are also indicated, using the symbol  $\_$  for an infinite (unbounded) rank, and with ranks dependent on the ranks of argument verbs shown as *mu*, *lv*, etc.

Examples are provided with each definition, and the more complex of them may use auxiliary functions as yet unfamiliar. These examples may be approached by ignoring all but the immediately relevant aspects of such auxiliaries, and by examining (and perhaps entering for execution) component phrases that can be used to build up the final result.

For example, in the discussion of the adverb  $\_$ , the sentences below display various uses of it for convenient comparison:

$$x=: 1\ 2\ 3\ 4\ 5\ [ \quad y=: 7\ 5\ 3$$

$$(\_ , \_ x) ; (x + / y) ; y ; (x * / y) ; (+ / y) ; (* / y)$$

1	8	6	4	7	5	3	7	5	3	15	105
2	9	7	5				14	10	6		
3	10	8	6				21	15	9		
4	11	9	7				28	20	12		
5	12	10	8				35	25	15		

Even if the auxiliary functions  $\_$  and  $\_ , \_$  are unfamiliar, their relevant effects are probably evident; if not, they may be clarified by the following experiments:

$x ; y$	$\_ , \_ 7\ 8$	$\$ , \_ 7\ 8$
7		
8		

Although a name (such as *foreign* for  $\_ ! :$ ) is suggested for each word, others can be used in addition to or instead of them. Thus, *joy* might be used for  $\_ !$  since the exclamation mark derives from an I placed above an o, an abbreviation of the Latin *io*. Similarly, *iota* might be used instead of *integers* and *index of* for  $\_ i$ .



# Self classify = \_ 0 0 Equal

`=y` classifies the items of the nub of `y` (that is, `~.y`) according to equality with the items of `y`, producing a boolean table of shape `#~.y` by `#y`. For example:

```
y=: 3 3 $ 'abcdef'
y ; (~.y) ; (=y)
+---+---+---+
|abc|abc|1 0 1|
|def|def|0 1 0|
|abc|    |    |
+---+---+---+
```

`x=y` is 1 if `x` is equal to `y`, and is otherwise 0.

The comparison is made with a *tolerance* `t`, normally 2 to the power `_44` but also controlled by the fit conjunction `!.`, as in `x=! .0 y`. Formally, `x=y` is 1 if the magnitude of `x-y` does not exceed `t` times the larger of the magnitudes of `x` and `y`.

Tolerance applies similarly to other verbs as indicated for each, notably to Match (`-:`), to Floor (`<.`), and to Signum (`*`), but not to Grade (`/:`).

Both the monadic and dyadic cases of the verb `=` apply to nouns of any rank, and to boxed as well as simple nouns. For example:

```
la=: ;: 'Try and try and try again.'
+---+---+---+---+---+
|Try|and|try|and|try|again.|
+---+---+---+---+---+

~. A
+---+---+---+
|Try|and|try|again.|
+---+---+---+

=a
1 0 0 0 0 0
0 1 0 1 0 0
0 0 1 0 1 0
0 0 0 0 0 1

a= <'and'
0 1 0 1 0 0
```

Because of the limited precision of the computer, results which should agree (such as `144*(13%144)` and `13`) may not; the tolerant comparison allows such a comparison to show agreement (a result 1). More or less stringent comparisons may be made by using the conjunction `!.` to specify a tolerance `t`, as in the function `eq=: =! .t`.

## Copula (Is)            = .            Local            = :            Global

The copula is used to assign a referent to a name, as in `a=:3` and in `sum=:+/.`. The copula `=.` is *local* as discussed under Explicit Definition (`:`), and `=:` is *global*. Copulas may also be used *indirectly*, with the name or names specified as a character list or a boxed list; moreover, if the character list begins with ``` then gerund referents are evoked.

For example:

```
f=: 3 : 0
a=. +:y.
b=: *:a
10*b
)

a=: b=: 678
a,b
678 678

f 3
360
a,b
678 36

x=: 'abc';'c'
(x) =: 3 4 ; 5 6 7
abc
3 4
c
5 6 7
```

Note that the parentheses around the name `x` force it to be evaluated before the assignment specified by the copula is effected.

```
'alpha beta'=: i.2 4
alpha
0 1 2 3
beta
4 5 6 7

'`sum sqrt'=: +/ ` %:
sum 3 1 4 2
10
sqrt 2
1.41421
```

**Box**

&lt; \_ 0 0

**Less Than**

<y is an <i>atomic encoding</i> of y, as discussed in Section II A. The result has rank 0, and is <i>decoded</i> by > .	x<y is 1 if x is tolerantly less than y. See Equal (=) for a definition of tolerance. <!.t uses tolerance t.
---	--

Boxing is also effected by verbs such as Link (;) and Word Formation (:):

```
(<'abc'),(<5 7),(<i.2 3)
+---+---+---+
|abc|5 7|0 1 2|
|   |   |3 4 5|
+---+---+---+

;: 'Now is the time'
+---+---+---+
|Now|is|the|time|
+---+---+---+

] a=: 2;3 5;7 11 13
+---+---+---+
|2|3 5|7 11 13|
+---+---+---+

>a
2 0 0
3 5 0
7 11 13
```

Cut (;.) with < has several uses (chosen by the right argument); the phrase <@v avoids the padding (and some domain errors) that may result from applying v alone:

```
<:._1 '/i sing/of olaf/'
+---+---+---+
|i sing|of olaf||
+---+---+---+

i."(0) 2 3 4
0 1 0 0
0 1 2 0
0 1 2 3

<@i."(0) 2 3 4
+---+---+---+
|0 1|0 1 2|0 1 2 3|
+---+---+---+
```

If y is a high-rank array, <"\_1 y or <"\_2 y often gives a more intelligible display than y itself. The display of a boxed array would normally be corrupted by control

characters (such as carriage returns and linefeeds) occurring therein; in the display such characters are replaced by spaces. For example, try `< 8 32 $ a.`

## Floor < . 0 0 0 Lesser Of (Min)

<code>&lt;.y</code> gives the <i>floor</i> of <code>y</code> , that is, the largest integer less than or equal to <code>y</code> . Thus: <code>&lt;. 4.6 4 _4 _4.6</code> <code>4 4 _4 _5</code> The implied comparison with integers is tolerant, as discussed under Equal (=), and is controlled by <code>&lt;.! .t .</code> . See below for complex arguments.	<code>x&lt;.y</code> is the lesser of <code>x</code> and <code>y</code> . For example: <code>3 &lt;. 4 _4</code> <code>3 _4</code> <code>&lt;./7 8 5 9 2</code> <code>2</code> <code>&lt;./\7 8 5 9 2</code> <code>7 7 5 5 2</code>
--	---

For a complex argument, the definition of `<.` is modelled by:

```
floor=: j./@(ip+(c2>c1),c1+:c2)
'`c1 c2 fp ip'=: (1:>+/@fp)`(>:/@fp)`(+.-ip)`(<.@+.)
```

As developed by McDonnell [10], this function has the following properties:

- Convexity: If  $(\langle .z_1) = (\langle .z_2)$  and  $z_3$  lies on the line between  $z_1$  to  $z_2$ , then  $(\langle z_3) = (\langle z_1)$ .
- Translatibility: If  $z_4$  is a Gaussian integer, then  $(z_4 + \langle .z_5) = (\langle .z_4 + z_5)$ .
- Compatibility:  $(\langle .x \ j.0) = ((\langle .x) j.0)$  and  $(\langle .0 \ j.x) = (0 \ j.(\langle .x))$

The function `<.` can be viewed as a tiling by rectangles of unit area, all arguments within a rectangle sharing the same floor. One rectangle has vertices at `1 j0` and `0 j1`, with the other side passing through the origin. Rectangles along successive diagonals are displaced by one-half the length.

The phrase `j./@ip` “floors” the individual parts of a complex argument. Moreover, the floor `<.y` is equivalent to `->.-y`. In other words, it is the dual of *ceiling* with respect to (that is, *under*) arithmetic negation: `<. ↔ >.&.-` and `>. ↔ <.&.-`. Thus:

```
(>.&.- ; <.) 4.6 4 _4 _4.6
+-----+
| 4 4 _4 _5 | 4 4 _4 _5 |
+-----+
```

Continued

**Floor**                      **< . 0 0 0**                      **Lesser Of (Min)**

Continued
-----------

The expression `<.x+0.5` gives the integer *nearest* to the real argument `x`, and `<.z+0.5j0.5` gives the Gaussian integer nearest to `z`. The number of digits needed to represent an integer is given by one plus the floor of its base ten logarithm:

```

a ,. (,. 1: +<.) 10^. a=: 9 10 11 99 100 101
9 0.954243 1
10      1 2
11 1.04139 2
99 1.99564 2
100      2 3
101 2.00432 3

```

**Decrement      <: 0 0 0      Less than or Equal**

<p>&lt;:y is y-1. For example:</p> <p>      &lt;: 2 3 5 7</p> <p>1 2 4 6</p> <p>Also see Not (-.)</p>	<p>x&lt;:y is 1 if x is less than or equal to y, and is otherwise 0. See Equal (=) for a discussion of tolerance. The fit conjunction (!.) applies to &lt;:.</p>
---	--

The inverse of <: is >: (Increment). For example:

n=: 5

<: ^: \_1 n

6

<: ^: 0 1 2 n

5 4 3

<: ^: i. n

5 4 3 2 1

\*/ <: ^: i. n

120

f=: \*/ @ (<: ^: i.)

f n

120

f"0 i. n

1 1 2 6 24

(f"0 = !) i. n

1 1 1 1 1

<:/ ~ i. 5

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

Here ^: applies to a noun right argument (0 1 2)

Here ^: applies to a verb right argument (i.)

Table of the dyad <:

# Open > 0 0 0 Larger Than

Open is the inverse of box, that is, $><y$ is $y$ . When applied to an open array (that has no boxed elements), open has no effect. Opened atoms are brought to a common shape as discussed in Sec. II B.	$x>y$ is 1 if $x$ is tolerantly larger than $y$ . See Equal (=) for a discussion of tolerance. For example: <pre> 1 2 3 4 5 &gt; 5 4 3 2 1 0 0 0 1 1 </pre> Tolerance $t$ is provided by $>!.t$ .
---	--

Since the rank of open is 0, it applies to each atom of its argument. For example:

```

]a=: 1 2 3;4 5 6;7 8 9
+-----+-----+-----+
| 1 2 3|4 5 6|7 8 9|
+-----+-----+-----+

>a
1 2 3
4 5 6
7 8 9

```

Results of different shapes are padded as defined in Section II B. For example:

```

(>1;2 3;4 5 6); (>'a';'bc';'def'); (<\i.4); (><\i.4)
+-----+-----+-----+-----+
| 1 0 0|a| +-+---+-----+-----+| 0 0 0 0| | | | | |
| 2 3 0|bc| |0|0 1|0 1 2|0 1 2 3| | 0 1 0 0|
| 4 5 6|def| +-+---+-----+-----+| 0 1 2 0|
|      |   | |0 1 2 3|
+-----+-----+-----+-----+

```

```

</~ i.5
0 1 1 1 1
0 0 1 1 1
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0

```

Table of the dyad  $<$ :

```

1 < 1+10^-8+i.15
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0

```

Tolerant comparison

```

1 <!. (0) 1+10^-8+i.15
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0

```

Exact comparison (0-tolerance)

Ceiling

> . 0 0 0

Larger Of (Max)

<p>&gt;.y gives the <i>ceiling</i> of y, that is, the smallest integer greater than or equal to y. Thus:</p> <pre>&gt;. 4.6 4 _4 _4.6 5 4 _4 _4</pre> <p>The implied comparison with integers is tolerant, as discussed under Equal (=), and is controlled by &gt;.! .t . See Floor (&lt;.) and McDonnell [10] for complex arguments.</p>	<p>x&gt;.y is the larger of x and y. For example:</p> <pre>3&gt;.4 _4 4 3  &gt;./7 8 5 9 2 9  &gt;./\7 8 5 9 2 7 8 8 9 9</pre>
---	--

The comparison `x = >. x` determines whether `x` is an integer. Thus:

```
Integer_test=: ] = >.
Integer_test 3 3.14 _5
1 0 1
```

See the definition of fork in Section II F.

```
f=: = >.
f 3 3.14 _5
1 0 1
```

The same function may be defined by a hook.

The *ceiling* `>. y` is equivalent to `-<.-y`. In other words, it is the dual of *floor* with respect to (that is, *under*) arithmetic negation: `>. ↔ <.&.-` and `<. ↔ >.&.-`. For example:

```
(<.&.- ; >.) 4.6 4 _4 _4.6
+-----+-----+
|5 4 _4 _4|5 4 _4 _4|
+-----+-----+
```



# Increment $> : 0 \ 0 \ 0$ Larger or Equal

$> : y$  is  $y+1$ . For example:

```
> : 2 3 5 7
3 4 6 8
```

Also see Not ( $\neg$ .)

$x > : y$  is 1 if  $x$  is tolerantly greater than or equal to  $y$ .

See Equal ( $=$ ) for a discussion of tolerance.

$> : !. t$  uses tolerance  $t$ .

```
+ : i. 6
0 2 4 6 8 10
```

Even numbers

```
> : + : i. 6
1 3 5 7 9 11
```

Odd numbers

```
odds = : > : @ + : @ i.
odds 10
1 3 5 7 9 11 13 15 17 19
```

```
+ / odds 10
100
```

```
(+ / @ odds , * :) 10
100 100
```

Sum of first  $n$  odds equals the square of  $n$

```
> : / ~ i. 5
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
```

Table of the dyad  $> :$

## Negative Sign and Infinity `_`

The symbol `_` followed by a digit denotes a negative number (as in `_3.4`), and denotes infinity when used alone, or negative infinity (in `__`). It is also used in names, as discussed in Part I and in Part II Section I.

For example:

<code>2 % 0</code>	Two divided by zero
<code>_</code>	
<code>10 ^. 0</code>	Base ten logarithm of zero
<code>__</code>	
<code>_2 _ 3 + 5</code>	
<code>3 _ 8</code>	
<code>integer_test=: =&lt;.</code>	Use of break in name
<code>integer_test 3 3.5</code>	
<code>1 0</code>	

Although `-2` may sometimes be used instead of `_2`, it is important to understand that the former is the application of a function to the number `2`, whereas the symbol `_` is an indivisible part of the number representation, just as the period is an indivisible part of a number such as `8.9`.

## Indeterminate $\frac{\infty}{\infty}$ .

The indeterminate  $\frac{\infty}{\infty}$  results from expressions such as  $\frac{\infty}{\infty}$  (infinity minus infinity) and from expressions (such as  $\frac{3}{\infty}$ ) in which an indeterminate argument occurs.

Infinity

\_\_ : \_\_ \_\_ \_\_

Infinity

<code>__:</code> is a <i>constant</i> function that yields an infinite result, that is, <code>__:</code> <code>y</code> is <code>__</code>	<code>__:</code> is a <i>constant</i> function that yields an infinite result, that is, <code>x __:</code> <code>y</code> is <code>__</code>
--	--

For example:

```
y=: 1 2 3 4
__ : y
_
_ "0 y      Rank zero applies to each element
_ _ _ _
```

Other constant functions include `__9:` and `__8:` etc. to `9:` . More generally, the expression `x"r` defines a constant function of rank `r` that yields the constant value `x` . For example:

```
3.14"0 y
3.14 3.14 3.14 3.14

3.14"1 y
3.14
```

The specific constant functions mentioned can therefore be written alternatively as `__"` and `__9"` and `0"` and `9"`, etc.

# Conjugate

+ 0 0 0

# Plus

+ y is the <i>conjugate</i> of y. For example, +3j4 is 3j_4.	+ is defined as in elementary arithmetic, and is extended to complex numbers as usual.
--	--

A complex number y multiplied by its conjugate produces a real number equal to the square of its magnitude |y|. For example:

$$\begin{array}{l} 3j4 * 3j\_4 \\ 25 \end{array}$$

The function j. multiplies its argument by the square root of negative one:

$$\begin{array}{l} ]i=: i. 5 \\ 0 1 2 3 4 \end{array}$$

$$\begin{array}{l} j. i \\ 0 0j1 0j2 0j3 0j4 \end{array}$$

$$\begin{array}{l} ]y=: i + 2 * j. i \\ 0 1j2 2j4 3j6 4j8 \end{array}$$

$$\begin{array}{l} +y \\ 0 1j\_2 2j\_4 3j\_6 4j\_8 \end{array}$$

$$\begin{array}{l} y * +y \\ 0 5 20 45 80 \end{array}$$

$$\begin{array}{l} \%: y * +y \\ 0 2.23607 4.47214 6.7082 8.94427 \end{array}$$

$$\begin{array}{l} |y \\ 0 2.23607 4.47214 6.7082 8.94427 \end{array}$$

The conjugate of y can also be expressed as (|y\*y)%y. For example:

$$\begin{array}{l} (|y*y)\%y \\ 0 1j\_2 2j\_4 3j\_6 4j\_8 \end{array}$$

## Real / Imaginary      + . 0 0 0      GCD (Or)

`+.y` yields a two-element list of the real and imaginary parts of its argument. For example, `+.3j5` is `3 5`, and `+.3` is `3 0`.

`x+.y` is the *greatest common divisor* of `x` and `y`. If the arguments are boolean (0 or 1), the functions `+.`  and `*.`  are equivalent to logical *or* and *and*. The function `-.`  similarly restricted is *not*.

```
l y=: i+2*j. i=: i.4
0 1j2 2j4 3j6
```

```
+. y
0 0
1 2
2 4
3 6
```

The greatest common divisor divides both of its arguments `x` and `y` to produce results that have no common factor, that is, the GCD of the quotients is 1. Moreover, these quotients represent the fraction `x%y` in lowest form. For example:

```
x=: 24 [ y=: 60
x;y;(x+.y);((x , y) % (x+.y))
+---+---+---+---+
|24|60|12|2 5|
+---+---+---+---+
```

```
lff=: , % +.      Gives lowest form of fraction
x;y;(x lff y);(%/x lff y);(%/x,y);(+./x lff y)
+---+---+---+---+---+
|24|60|2 5|0.4|0.4|1|
+---+---+---+---+---+
```

Since the functions `=|` and `=<.` (tests for non-negative and for integer) produce boolean results, the phrase `(=|)+.(=<.)` is a test for non-negative *or* integer:

```
(test=: (=|)+.(=<.)) _2 _2.4 3 3.5
1 0 1 1
```

The duality of *or* and *and* may be shown as follows:

```
d (+./ ; *.&.-./ ; *./ ; +.&.-./) d=: 0 1
+---+---+---+---+
|0 1|0 1|0 0|0 0|
|1 1|1 1|0 1|0 1|
+---+---+---+---+
```

**Double****+ : 0 0 0****Not-Or**

+: y is twice y. For example:

```

+ : 3 0 _2
6 0 _4

```

x +: y is the negation of x *or* y. For example, 0 +: 0 is 1.

Since the square of the sum of two arguments equals the sum of their squares and *twice* their product, the following functions are equivalent:

```

f=: + * +
g=: *:@[ + +:@* + *:@]

```

For example:

```

x=: 7 6 3 [ y=: 6 5 3
x (f ; g ; (f=g) ; (f-:g)) y
+-----+-----+-----+-----+
|169 121 36|169 121 36|1 1 1|1|
+-----+-----+-----+-----+

```

Since the domain of not-or is limited to zero and one, its entire behaviour can be seen in the following function tables:

```

d=: 0 1
d +:/ d
1 0
0 0

```

Domain of nor  
Table of nor

```

d +./ d
0 1
1 1

```

Table of or

```

-. d +./ d
1 0
0 0

```

Negation of table of or

```

(+:&.-./~d) ; (*:/~d)
+----+----+
|1 1|1 1|
|1 0|1 0|
+----+----+

```

Nand and nor are duals under not

Signum

\* 0 0 0

Times

<p><code>*y</code> is <code>_1</code> if <code>y</code> is negative, <code>0</code> if it is zero, <code>1</code> if it is positive; more generally, <code>*y</code> is the intersection of the unit circle with the line from the origin through the argument <code>y</code> in the complex plane. For example:</p> <pre>      *_3 0 5 3j4 _1 0 1 0.6j0.8</pre> <p>The comparison with zero is tolerant, as defined by the phrase <code>(y% y)*t&lt;: y</code> where <code>t</code> denotes the tolerance. The fit conjunction applies to signum, as in <code>*!.t.</code></p>	<p><code>*</code> denotes multiplication, defined as in elementary mathematics and extended to complex numbers as usual:</p> <pre>t=:+.x,y [ x=:2j4 [ y=:5j3 r=:-/*/t [ i=:+/ . * t (x,:y);t;r;i;(r j. i);(x*y)</pre> <table><tr><td>2j4</td><td>2 4</td><td>-2</td><td>26</td><td>-2j26</td><td>-2j26</td></tr><tr><td>5j3</td><td>5 3</td><td></td><td></td><td></td><td></td></tr></table>	2j4	2 4	-2	26	-2j26	-2j26	5j3	5 3				
2j4	2 4	-2	26	-2j26	-2j26								
5j3	5 3												

Signum is useful in effecting selections. For example:

```
      * y=: _4 0 4
      _1 0 1

      >:@* y
      0 1 2

      f=: %:
      f ^: * " 0 y
      16 0 2
```

Inverse of `f`, Identity, or `f`

```
      (* y) { ;:'Yes No Maybe'
```

Select using indexing (`{`)

Maybe	Yes	No
-------	-----	----

```
      g=: <:~ -:~+:@.*"0
      g y
      _8 _1 2
```

See Agenda (`@.`)

The dyad `*` used on a list and a table illustrates the significance of *agreement*, as discussed in Section II B:

<code>m=: i. 3 4 [ v=: 3 2 1</code>				<code>m ; (v*m) ; (m*v) ; (+/ m*v) ; (v +/ . * m)</code>			
0 1 2 3	0 3 6 9	0 3 6 9	16 22 28 34	16 22 28 34			
4 5 6 7	8 10 12 14	8 10 12 14					
8 9 10 11	8 9 10 11	8 9 10 11					



+-----+-----+-----+-----+-----+

**Length / Angle      \* . 0 0 0      LCM (And)**

<code>*.y</code> is a two-element list of the length and angle (in radians) of the hypotenuse of a triangle with base and altitude equal to the real and imaginary parts <code>y</code> . For example, <code>*. 3j4</code> is <code>5 0.927295</code> .	<code>x*.y</code> is the least common multiple of <code>x</code> and <code>y</code> . For boolean arguments ( <code>0</code> and <code>1</code> ) it is equivalent to <i>and</i> . Thus:  0 1 *. / 0 1 0 0 0 1
---	--

Some properties of the length / angle are illustrated in the following, including the fact that the length (i.e. magnitude) of the product of two complex numbers is the product of their lengths, and the angle of the product is the sum of their angles:

```
(| ; *. ; r./@*. ) y=: 3j4
+---+-----+-----+
|5|5 0.927295|3j4|
+---+-----+-----+

x=: 2j_6
*. x,y
6.32456 _1.24905
5 0.927295

f=: */@:( { ."1 ) , +/@:( { ."1 )
f *. x , y
31.6228 _0.321751

*. x * y
31.6228 _0.321751
```

Polar coordinates

Product over first col and sum over last

Length and angle of product

The least common multiple is the product divided by the GCD. For example:

```
24 (+. ; *. ; */ % +.) 60
+---+-----+-----+
|12|120|120|
+---+-----+-----+
```

**Square****\* : 0 0 0****Not-And**

\* : y is the square of y.

x \* : y is the negation of x *and* y. For example 0 \* : 0 is 1.

The inverse of the square is the square root. For example:

```
*: ^: _1 (_2 _1 0 1 2)
0j1.41421 0j1 0 1 1.41421
```

```
3 +&.*: 4
5
```

Hypotenuse of triangle with sides 3 and 4

Since the domain of nand is limited to zero and one, its entire behaviour can be seen in the following function tables:

```
d=: 0 1
d *: / d
```

Domain of nand  
Table of nand

```
1 1
1 0
```

```
d *./ d
```

Table of and

```
0 0
0 1
```

```
-. d *./ d
```

```
1 1
1 0
```

Nand, Not and, and the dual of Nor all agree, as illustrated below:

```
( *: / ~ ; -. @ * . / ~ ; + : & . - . / ~ ) d
```

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

Negate

-

0

0

0

Minus

<div><div><div>-y</div> is the negative of <div>y</div>. That is, it is defined as <div>0 - y</div>. Thus, <div>-2 0 _2</div> is <div>_2 0 2</div>.</div></div>	<div><div>-</div> is defined as in elementary arithmetic, and is extended to complex numbers as usual.</div>
---	--

The function 

-

 is self-inverse, that is, 

-^:\_1

 is 

-

 itself.

Although 

-2

 may often be used instead of 

\_2

, it is important to understand that the former is the application of a function to the number 

2

, whereas the symbol 

\_

 is an indivisible part of the number representation, just as the period is an indivisible part of a number such as 

8.9

.

**Not**

$$- . 0 \_ \_$$
**Less**

$- . y$  is  $1-y$ ; for a boolean argument it is the complement (not); for a probability, it is the complementary probability.

$x - . y$  includes all items of  $x$  except for those that are cells of  $y$ .

Tolerance  $\tau$  is provided by  $- . ! . \tau$ .

The function *less* applies to any conformable pair of arguments. For example:

```
(i. 9) -. 2 3 5 7
0 1 4 6 8
```

```
'abcdefghij' -. 'aeiou'
bcd fghj
```

```
]m=: i. 4 5
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

```
m -. 5 6 7 8 9
0 1 2 3 4
10 11 12 13 14
15 16 17 18 19
```

```
b=: <\ 'abcdefg'
b
+---+
|a|ab|abc|abcd|abcde|abcdef|abcdefg|
+---+
```

```
b -. 'abc'; 'abcde'; 'cba'
+---+
|a|ab|abcd|abcdef|abcdefg|
+---+
```

```
2 3 4 5 -. 'abcdef'
2 3 4 5
```

Halve

- : 0 \_ \_

Match

<div>-:y is one half of y. For example: <div>-: i. 5 0 0.5 1 1.5 2</div></div>	<div>x -: y yields 1 if its arguments match: in shapes, boxing, and elements; but using tolerant comparison. See Equal (=).  Matching with a tolerance t can be obtained using the verb -: !. t.</div>
--	--

For example:

```
x=: 0 1 2 3 4 5
,.&.> (] ; -: ; +:@-: ; (%&2) ; (2: %~ ])) x

+---+---+---+
|0| 0|0| 0| 0|
|1|0.5|1|0.5|0.5|
|2| 1|2| 1| 1|
|3|1.5|3|1.5|1.5|
|4| 2|4| 2| 2|
|5|2.5|5|2.5|2.5|
+---+---+---+

x = +: -: x
1 1 1 1 1 1

x -: +: -: x
1
```

# Reciprocal                      % 0 0 0                      Divided by

% y is the reciprocal of y, that is, 1/y. For example, %4 $\leftrightarrow$ 0.25.	x % y is division of x by y as defined in elementary math, except that 0%0 is 0. See McDonnell [11], and the resulting pattern in the middle column and middle row of the table below.
--	---

We will illustrate the divide function by tables, using a function to generate lists symmetric about zero:

```
sym=: i.@>:@+:- ]          Symmetric integers
] a=: sym 3
_3 _2 _1 0 1 2 3
(] ; *) |. a%/a
```

	_1	_1.5	_3	—	3	1.5	1		_1	_1	_1	1	1	1	1
_0.666667		_1	_2	—	2	1	0.666667		_1	_1	_1	1	1	1	1
_0.333333		_0.5	_1	—	1	0.5	0.333333		_1	_1	_1	1	1	1	1
	0	0	0	0	0	0		0	0	0	0	0	0	0	0
0.333333	0.5	1	—	_1	_0.5	_0.333333		1	1	1	_1	_1	_1	_1	_1
0.666667	1	2	—	_2	_1	_0.666667		1	1	1	_1	_1	_1	_1	_1
	1	1.5	3	—	_3	_1.5	_1		1	1	1	_1	_1	_1	_1

```
6j2 ": |. a %/ a
_1.00 _1.50 _3.00      — 3.00 1.50 1.00
_0.67 _1.00 _2.00      — 2.00 1.00 0.67
_0.33 _0.50 _1.00      — 1.00 0.50 0.33
0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.33 0.50 1.00      — _1.00 _0.50 _0.33
0.67 1.00 2.00      — _2.00 _1.00 _0.67
1.00 1.50 3.00      — _3.00 _1.50 _1.00
```

The final use of the format function gives a more readable result, with a width of six spaces per column and a uniform two digits after the decimal point.

```
|. a %/ x: a
_1 _3r2 _3 — 3 3r2 1
_2r3 _1 _2 — 2 1 2r3
_1r3 _1r2 _1 — 1 1r2 1r3
0 0 0 0 0 0 0
1r3 1r2 1 — _1 _1r2 _1r3
2r3 1 2 — _2 _1 _2r3
1 3r2 3 — _3 _3r2 _1
```

## Matrix Inverse $\% . \ 2 \ \_ \ 2$ Matrix Divide

If  $y$  is a non-singular matrix, then  $\% . y$  is the inverse of  $y$ . For example:

```
mp=: +/ . *      Matrix product
(% . ; ] ; % . mp ] ) i. 2 2
+-----+-----+
|_1.5 0.5|0 1|1 0|
|  1   0|2 3|0 1|
+-----+-----+
```

More generally,  $\% . y$  is defined in terms of the dyadic case, with the left argument  $= i. \{ : \$ y$  (an identity matrix) or, equally, by the relation  $(\% . y) mp \ x \leftrightarrow x \% . \ y$ .

The shape of  $\% . y$  is  $| . \$ y$ .

The vector and scalar cases are defined by using the matrix  $, . y$ , but the shape of the result is  $\$ y$ .

For a non-zero vector  $y$ , the result of  $\% . y$  is a vector collinear with  $y$  whose length is the reciprocal of that of  $y$ ; it is called the reflection of  $y$  in the unit circle (or sphere). Thus:

```
(% . ,: ] % % .) 2 3 4
0.0689655 0.103448 0.137931
      29      29      29
```

If  $y$  is non-singular, then  $x \% . y$  is  $(\% . y) mp \ x$ . More generally, if the columns of  $y$  are linearly independent and if  $\#x$  and  $\#y$  agree, then  $x \% . y$  minimizes the difference:

$$d = \| x - y mp \ x \% . y \|$$

in the sense that the magnitudes  $\|d\|$  are minimized. Scalar and vector cases of  $y$  are treated as the one-column matrix  $, . y$ .

Geometrically,  $y mp \ x \% . y$  is the *projection* of the vector  $x$  on the column space of  $y$ , the point nearest to  $x$  in the space spanned by the columns of  $y$ .

Common uses of  $\% .$  are in the solution of linear equations and in the approximation of functions by polynomials, as in  $c = (f \ x) \% . \ x ^ / i. 4$ .

We will illustrate the use of  $\% .$  in function fitting by the sine function, showing, in particular, the maximum over the magnitudes of the differences from the function being approximated:

```
sin=: 1&o. [ . x=: 5 %~ i. 6      Function to be approximated
c=: (sin x) % . x ^ / i. 4      Use of matrix divide
, .&.>@[ ; c"_ ; sin ; c&p. ; >./@:|@(sin-c&p.)) x
+-----+-----+-----+-----+
| 0 |_5.30503e_5| 0 |_5.30503e_5| 0.000167992|
| 0.2 | 1.00384 | 0.198669 | 0.198826 |
| 0.4 |_0.018453 | 0.389418 | 0.389321 |
| 0.6 |_0.143922 | 0.564642 | 0.564523 |
| 0.8 | 0.717356 | 0.717524 |
| 1 | 0.841471 | 0.841416 |
+-----+-----+-----+-----+
```



**Square Root****% :** 0 0 0**Root**

%: y is the square root of y. If y is negative, the result is an imaginary number. For example, %:-4 $\leftrightarrow$ 0j2.	x %: y is the x root of y. Thus, 3%:8 is 2, and 2%:y is %:y. In general, x %: y $\leftrightarrow$ y^%x.
---	---

For example:

```

y=: i. 7
Y
0 1 2 3 4 5 6

2 %: y
0 1 1.41421 1.73205 2 2.23607 2.44949

%: y
0 1 1.41421 1.73205 2 2.23607 2.44949

r=: 1 2 3 4
z=: r %:/ y
z
0 1      2      3      4      5      6
0 1 1.41421 1.73205      2 2.23607 2.44949
0 1 1.25992 1.44225 1.5874 1.70998 1.81712
0 1 1.18921 1.31607 1.41421 1.49535 1.56508

r ^~ z
0 1 2 3 4 5 6
0 1 2 3 4 5 6
0 1 2 3 4 5 6
0 1 2 3 4 5 6

```

See *agreement* in Section II B, and note use of ~

# Exponential

^ 0 0 0

# Power

$x^y$  is equivalent to  $e^{y \ln x}$ , where  $e$  is Euler's number  $e^1$  (approximately 2.71828). The *natural logarithm* ( $\ln$ ) is inverse to  $e^x$  (that is,  $y = \ln x$  and  $x = e^y$ ).

The monad  $x \&^$  is inverse to the monad  $x \&^.$ . For example:

```
10&^ 10&^. 1 2 3 4 5
1 2 3 4 5
```

```
10&^. 10&^ 1 2 3 4 5
1 2 3 4 5
```

$x^2$  and  $x^3$  and  $x^{0.5}$  are the *square*, *cube*, and *square root* of  $x$ .

In general,  $x^y$  is  $y^{x^y}$ , applying for complex numbers as well as real.

For a non-negative integer  $y$ , the phrase  $x \&^ y$  is equivalent to  $x^y$ ; in particular,  $x \&^$  on an empty list is 1, and  $x \&^ 0$  is 1 for any  $x$ , including 0.

The fit conjunction applies to  $\&^$  to yield a *stope* defined as follows:  $x! \cdot k \cdot n$  is  $x^k / x + k \cdot i \cdot n$ . In particular,  $\&^! \cdot 1$  is the *falling factorial* function.

The last result in the first example below illustrates the falling factorial function, formed by the fit conjunction. See Chapter 4 of [8] for the use of *stope* functions, *stope* polynomials, and Stirling numbers in the difference calculus:

```
e=: ^ 1 [ x=: 4 [ y=: 0 1 2 3
,.&.> x (e"_ i e&^@] i ^ i ^@([ * ^.@]) i ([^]) i ^!._1) y
```

2.71828	1	1	1	1
2.71828	4	1	1	4
7.38906	16	4	4	12
20.0855	64	27	27	24

```
S2=: %.@S1=: (^!._1/~ %.^/~) @ i. @ x:
(S1;S2) 8
```

1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	1	2	6	24	120	720	0	1	1	1	1	1	1	1	1
0	0	1	3	11	50	274	1764	0	0	1	3	7	15	31	63	63
0	0	0	1	6	35	225	1624	0	0	0	1	6	25	90	301	301
0	0	0	0	1	10	85	735	0	0	0	0	1	10	65	350	350
0	0	0	0	0	1	15	175	0	0	0	0	0	1	15	140	140
0	0	0	0	0	0	1	21	0	0	0	0	0	0	1	21	21
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1

$S1$  gives (signed) Stirling numbers of the first kind and  $S2$  gives Stirling numbers of the second kind. They can be used to transform between ordinary and *stope* polynomials. Note that  $x:$  gives extended precision.

# Natural Log $\ln$ Logarithm

The <i>natural logarithm</i> $\ln(\cdot)$ is inverse to the exponential $e^x$ (i.e., $y = e^{\ln y}$ and $y = e^{\ln y}$ ).	The <i>base-x logarithm</i> $\log_x(\cdot)$ is the inverse of power $x^y$ in the sense that $y = x^{\log_x y}$ and $y = x^{\log_x y}$ .
---	---

Certain properties of logarithms are illustrated below:

```

x=: 4 [ y=: 0 1 2 3
(x^y) ; (x^.x^y) ; (x^.y) ; (x^x^y)
+-----+
| 1 4 16 64 | 0 1 2 3 | 0 0.5 0.792481 | 1 2 3 |
+-----+

logtable=: ^./~@i.
<6j2 ": logtable 6
+-----+
|   .  0.00  0.00  0.00  0.00  0.00 |
|   .  0.00  0.00  0.00  0.00  0.00 |
|   .  0.00  1.00  1.58  2.00  2.32 |
|   .  0.00  0.63  1.00  1.26  1.46 |
|   .  0.00  0.50  0.79  1.00  1.16 |
|   .  0.00  0.43  0.68  0.86  1.00 |
+-----+

```

The first derivative of the natural logarithm is the reciprocal. For example:

```

^ . d. 1 y=: 0 1 2 3 4 5 6
_ 1 0.5 0.333333 0.25 0.2 0.166667

% ^ . d. 1 y
0 1 2 3 4 5 6

```

## Power $u^n$ \_ \_ \_

Two cases occur: a numeric integer  $n$ , and a gerund  $n$ .

**Numeric case.** The verb  $u$  (or  $x&u$ ) is applied  $n$  times. For example:

( ] ; +/\ ; +/\^:2 ; +/\^:0 1 2 3 _1 _2 _3 _4 ) 1 2 3 4 5																			
1	2	3	4	5	1	3	6	10	15	1	4	10	20	35	1	2	3	4	5
										1	2	3	4	5					
										1	3	6	10	15					
										1	4	10	20	35					
										1	5	15	35	70					
										1	1	1	1	1					
										1	0	0	0	0					
										1	_1	0	0	0					
										1	_2	1	0	0					

An infinite power  $n$  produces the limit of the application of  $u$ . For example, if  $x=:2$  and  $y=:1$ , then  $x \circ.^:_ y$  is 0.73908, the solution of the equation  $y=\text{Cos } y$ . If  $n$  is negative, the obverse  $u^:_1$  is applied  $|n|$  times. The obverse (which is normally the inverse) is specified for six cases:

- The self-inverse functions  $+ \ - \ - . \ \% \ \% . \ | . \ | : \ / : \ [ \ ] \ C . \ p .$
- The pairs in the following lists:  
 $< \ < : \ + . \ + : \ + \sim \ * . \ * : \ * \sim \ ^ \ , : \ , \sim$   
 $> \ > : \ j . / " 1 " _ \ - : \ - : \ r . / " 1 " _ \ \% : \ \% : \ ^ . \ \{ . \ ( < . @ - : @ \# ) \& \{ .$   
 $i : \ \# . \ " . \ ; \sim \ 3 ! : 1 \ 3 ! : 3 \ p : \ q :$   
 $i @ ( , \& ' ' \& . > " 1 ) \ \# : \ " : \ > @ \{ . \ 3 ! : 2 \ 3 ! : 2 \ \pi (n) \ * /$   
 $\backslash : \ o . \ j . \ r .$   
 $/ : @ | . \ \% \& ( o . 1 ) \ \% \& 0 j 1 \ \% \& 0 j 1 @ ^ .$
- Obviously invertible bonded dyads such as  $- \& 3$  and  $1 0 \& ^ .$  and  $1 \ 0 \ 2 \& | :$  and  $3 \& | .$  and  $1 \& o .$  and  $a . \& i .$  as well as  $u @ v$  and  $u \& v$  if  $u$  and  $v$  are invertible.
- Monads of the form  $v / \backslash$  and  $v / \backslash .$  where  $v$  is one of  $+ \ * \ - \ \% \ = \ \sim :$
- Obverses specified by the conjunction  $: .$

**Continued**

## Power

$u^{\wedge} : n$                               

Continued
-----------

6. The following cases merit special mention:

$p^{\wedge} : \_1$   $n$  gives the number of primes less than  $n$ , denoted by  $\pi(n)$  in math

$q^{\wedge} : \_1$  is  $*/$

$\#^{\wedge} : \_1$  with a boolean left argument is “Expand” (whose fill atom  $f$  can be specified by *fit*,  $b\&\#^{\wedge} : \_1 ! . f$ )

$a\&\#^{\wedge} : \_1$  produces the base- $a$  representation

$!^{\wedge} : \_1$  and  $!\&n^{\wedge} : \_1$  and  $!\&n\&^{\wedge} : \_1$  produce the appropriate results

**Gerund case.** (Compare with the gerund case of the merge adverb  $\}$ )

$$\begin{aligned} x\ u^{\wedge} : (v0\`v1\`v2)\ y &\leftrightarrow (x\ v0\ y)\ u^{\wedge} : (x\ v1\ y)\ (x\ v2\ y) \\ x\ u^{\wedge} : (\quad\ v1\`v2)\ y &\leftrightarrow x\ u^{\wedge} : ([\`v1\`v2)\ y \\ u^{\wedge} : (\quad\ v1\`v2)\ y &\leftrightarrow u^{\wedge} : (v1\ y)\ (v2\ y) \end{aligned}$$

## Power $u^{\wedge} : v$

The case of  $^{\wedge} :$  with a verb right argument is defined in terms of the noun right argument case ( $u^{\wedge} : n$ ) as follows:

$$\begin{aligned} x \ u^{\wedge} : v \ y &\leftrightarrow x \ u^{\wedge} : (x \ v \ y) \ y \\ u^{\wedge} : v \ y &\leftrightarrow u^{\wedge} : (v \ y) \ y \end{aligned}$$

For example:

```
x=: 1 3 3 1
y=: 0 1 2 3 4 5 6
x p. y
1 8 27 64 125 216 343

x p. ^: (|>3:) "1 0 y
0 1 2 3 125 216 343

a=: _3 _2 _1 0 1 2 3
%: a
0j1.73205 0j1.41421 0j1 0 1 1.41421 1.73205

* a
_1 _1 _1 0 1 1 1

%: ^: * " 0 a
9 4 1 0 1 1.41421 1.73205

*: a
9 4 1 0 1 4 9
```

The following monads are equivalent. (See the example of  $^{\wedge} T.$       in the definition of the Taylor Approximation  $T.$      )

```
g=: u ^: p ^: _
h=: 3 : 't=. y. while. p t do. t=. u t end.'

u=: -&3 [. p=: 0&<
(g"0 ; h"0) i. 10
```

0	_2	_1	0	_2	_1	0	_2	_1	0	0	_2	_1	0	_2	_1	0	_2	_1	0
---	----	----	---	----	----	---	----	----	---	---	----	----	---	----	----	---	----	----	---

# Shape Of

\$ \_ 1 \_

# Shape

\$ y yields the shape of y as defined in Section II A. For example, the shape of a 2-by-3 matrix is 2 3, and the shape of the scalar 3 is an empty list (whose shape is 0).

The rank of an argument y is #@ \$ y.  
For example:

```
rank=: #@ $
(rank 3) , (rank ,3)
0 1
(rank 3 4),(rank i. 2 3 4)
1 3
```

The shape of x\$y is x,siy where siy is the shape of an item of y; x\$y gives a length error if y is empty and x,siy does not contain a zero. For example:

```
y=: 3 4$'abcdefghijkl'
y ; 2 2$ y
```

```
+-----+-----+
|abcd|abcd|
|efgh|efgh|
|ijkl|ijkl|
|abcd|abcd|
+-----+-----+
```

This example shows how the result is formed from the *items* of y, the last 1-cell (abcd) showing that the selection is cyclic. The fit conjunction (\$! . f) provides fill specified by the items of f.

Since x \$ y uses *items* from y, it is sometimes useful to ravel the right argument, as in x \$ ,y. For example (using the y defined above):

```
2 3 $ ,y
abc
def
```

The fit conjunction is often useful for appending zeros or spaces. For example:

```
8 $!.0 (2 3 4)
2 3 4 0 0 0 0 0
]z=: 8$!.'*' 'abc'
abc*****
|. z
*****cba

2 5$!.a: ;: 'zero one two three four five six'
+-----+-----+-----+-----+
|zero|one|two|three|four|
+-----+-----+-----+-----+
|five|six|   |   |   |
+-----+-----+-----+-----+
```

## Self Reference

\$ : \_ \_ \_

\$ : is a proxy that assumes the result of the phrase in which it occurs, the phrase being terminated on the left by a copula or by the completion of the sentence. For example:

```
1: `( ] * $:@<: )@. * 5
120
```

In the foregoing expression, the agenda (@.) chooses the verb ] \* \$:@<: as long as the argument (reduced by one each time by the application of the decrement) remains non-zero. When the argument becomes zero, the result of the right argument of @. is zero, and the constant function 1: is chosen.

If \$:@ were omitted from the expression, it would execute once only as follows:

```
1: `( ] * <: )@. * 5
20
```

The inclusion of self-reference ensures that the entire function is re-executed after decrementing the argument.



## Evolve

$m \sim \_$

If  $m$  is a name, then  $'m' \sim$  is equivalent to  $m$ . For example:

$m = : 2 \ 3 \ 4$   
 $'m' \sim$

$2 \ 3 \ 4$

$m = : +/$   
 $'m' \sim 2 \ 3 \ 5 \ 7$   
 17

$m = : /$   
 $+ 'm' \sim 2 \ 3 \ 5 \ 7$   
 17

Reflexive

$u \sim \_ ru \ lu$

Passive

$u \sim y$ is $y \ u \ y$ . For example, $\wedge \sim 3$ is 27, and $+/\sim i. \ n$ is an addition table.	$\sim$ <i>commutes</i> or <i>crosses</i> connections to arguments: $x \ u \sim \ y \leftrightarrow y \ u \ x$ .
---	---

Certain uses of the reflexive and passive are illustrated below:

```
x=: 1 2 3 4 [ y=: 4 5 6
x (.,.@[ i ^/ i ^/~ i ^/~@[ i ]) y

+-----+-----+-----+
| 1 | 1 | 1 | 1 | 4 16 64 256 | 1 | 1 | 1 | 1 | 4 5 6 |
| 2 | 16 | 32 | 64 | 5 25 125 625 | 2 | 4 | 8 | 16 |
| 3 | 81 | 243 | 729 | 6 36 216 1296 | 3 | 9 | 27 | 81 |
| 4 | 256 | 1024 | 4096 | 4 16 64 256 | 4 | 16 | 64 | 256 |
+-----+-----+-----+

into=: %~
(i. 6) % 5
0 0.2 0.4 0.6 0.8 1

5 into i. 6
0 0.2 0.4 0.6 0.8 1

from=: --~
(i.6) - 5
_5 _4 _3 _2 _1 0

5 from i.6
_5 _4 _3 _2 _1 0

(x %/ y);(x %~/ y);(x %/~ y)

+-----+-----+-----+
| 0.25 0.2 0.166667 | 4 | 5 | 6 | 4 | 2 1.33333 | 1 |
| 0.5 0.4 0.333333 | 2 | 2.5 | 3 | 5 | 2.5 1.66667 | 1.25 |
| 0.75 0.6 0.5 | 1.33333 1.66667 | 2 | 6 | 3 | 2 | 1.5 |
| 1 0.8 0.666667 | 1 | 1.25 1.5 |
+-----+-----+-----+
```

## Nub

~ . \_

`~.y` selects the *nub* of `y`, that is, all of its distinct items. For example:

```
y=: 3 3 $ 'ABCABCDEF'
y;(~.y);(~.3);($~.3)
```

ABC	ABC	3	1
ABC	DEF		
DEF			

More precisely, the nub is found by selecting the leading item, suppressing from the argument all items tolerantly equal to it, selecting the next remaining item, and so on. The fit conjunction applies to nub to specify the tolerance used.

If `f` is a costly function, it may be quicker to evaluate `f y` by first evaluating `f~. y` (which yields all of the distinct results required), and then distributing them to their appropriate positions. The inner product with the self-classification table (produced by `=`) can be used to effect this distribution. For example:

```
f=: *
f y=: 2 7 1 8 2 8 1 8
4 49 1 64 4 64 1 64
```

```
,.&.>(~. ; f@~. ; = ; (f@~.(+/ .*)=) ; f)y
```

2	4	1	0	0	0	1	0	0	0	4	4
7	49	0	1	0	0	0	0	0	0	49	49
1	1	0	0	1	0	0	0	1	0	1	1
8	64	0	0	0	1	0	1	0	1	64	64
										4	4
										64	64
										1	1
										64	64

```
NUB=: 1 : 'x.@~. +/ . * ='
*: NUB y
4 49 1 64 4 64 1 64
```

Adverb

```
nubindex=: ~. i. ]
(nubindex ; (nubindex { ~.)) y
```

0	1	2	3	0	3	2	3	2	7	1	8	2	8	1	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Nub Sieve

$\sim :$

$\_$

0

0

Not Equal

$\sim : y$ is the boolean list $b$ such that $b \# y$ is the nub of $y$ . For example:  $\sim : 'Mississippi'$ 1 1 1 0 0 0 0 0 1 0 0	$x \sim : y$ is 1 if $x$ is tolerantly unequal to $y$ . See Equal (=).  The fit conjunction may be used to specify tolerance, as in $\sim : ! . t$ .
---	--

The result of nub-sieve can be used to select the nub as follows:

```
y=: 8 1 8 2 8 1 7 2
~. y
8 1 2 7

~: y
1 1 0 1 0 0 1 0

(~: y) # y
8 1 2 7

y #~ ~: y
8 1 2 7
```

The dyad  $\sim :$  applies to any argument, but for booleans it is called *exclusive-or*. For example:

```
d=: 0 1
d ~:/ d
0 1
1 0
```

Not-equal, not equal, and the dual of equal with respect to not, all agree as illustrated below.

```
(~:/ ; -./ ; @= / ; =&.-./)~ d
+---+---+---+
|0 1|0 1|0 1|
|1 0|1 0|1 0|
+---+---+---+
```

# Magnitude | 0 0 0 Residue

$ y \leftrightarrow \%:y*+y$ . For example: $\begin{array}{r}   \ 6 \ _6 \ 3j4 \\ 6 \ 6 \ 5 \end{array}$	The familiar use of residue is in determining the remainder on dividing a non-negative integer by a positive: $\begin{array}{r} 3 \   \ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \\ 0 \ 1 \ 2 \ 0 \ 1 \ 2 \ 0 \ 1 \end{array}$
---	--

The definition  $y-x* < . \ y \% \ x+0=x$  extends the residue to a zero left argument, and to negative and fractional arguments. For example:

```
over =: ({ . ,.@; }.)@":@,
by   =: ' '&;@,.@[ ,. ]
x=: 3 2 1 0 _1 _2 _3 [ y=: 0 1 2 3 4 5 6 7 8
x by y over x | / y
```

	0 1 2 3 4 5 6 7 8
3	0 1 2 0 1 2 0 1 2
2	0 1 0 1 0 1 0 1 0
1	0 0 0 0 0 0 0 0 0
0	0 1 2 3 4 5 6 7 8
_1	0 0 0 0 0 0 0 0 0
_2	0 _1 0 _1 0 _1 0 _1 0
_3	0 _2 _1 0 _2 _1 0 _2 _1

To produce a true zero for cases such as  $(\%3) | (2\%3)$  the residue is made tolerant as shown in the definition of `res` below:

```
res=: f`g@.agenda"0
agenda=: ([ = 0:) +. (<. = >.)@S
S=: ] % [ + [ = 0:
f=: ] - [ * <.@S [ , g=: ] * [ = 0:
0.1 res 2.5 3.64 2 _1.6
0 0.04 0 0
```

Continued

Magnitude

000

Residue

Continued

(. . . i res/~ i |/~) a=: 2 ~ i.5

2	0	1	0	1	0	0	1	0	1	0
1	0	0	0	0	0	0	0	0	0	0
0	2	1	0	1	2	2	1	0	1	2
1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	1	0	0	1	0	1	0

The dyad `|` applies to complex numbers. Moreover, the fit conjunction may be applied to control the tolerance used. The dyad `m&|@^` on integer arguments is computed in a way that avoids large intermediate numbers. For example: `2 (1e6&|@^) 10^100x`

## Reverse

| . \_ 1 \_

## Rotate (Shift)

| . y reverses the order of the items of y. For example:

```
|. t=: 'abcdefg'
gfedcba
```

The *right shift* is the dyadic case of |.!.f with the left argument \_1. For example:

```
|.!. '#' t
#abcdef

|.!.10 i.3 3
10 10 10
0 1 2
3 4 5
```

x|.y rotates successive axes of y by successive elements of x. Thus:

```
1 2 |. i. 3 5
7 8 9 5 6
12 13 14 10 11
2 3 4 0 1
```

The phrase x|.!.f y produces a *shift*: the items normally brought around by the cyclic rotation are replaced by f unless f is empty (0=#f), in which case they are replaced by the normal fill defined under {.(take):

```
2 _2 |.!. '#' "0 1 t
cdefg##
##abcde
```

```
y=: a.{~ (a. i. 'A') + i. 5 6
```

```
(] ; 2&|. ; _2&|. ; 2&|. "1 ; 2&(|.!. '*' "1)) y
```

ABCDEF	MNOPQR	STUVWX	CDEFAB	CDEF**
GHIJKL	STUVWX	YZ[\]^	IJKLGH	IJKL**
MNOPQR	YZ[\]^	ABCDEF	OPQRMN	OPQR**
STUVWX	ABCDEF	GHIJKL	UVWXST	UVWX**
YZ[\]^	GHIJKL	MNOPQR	[\]^YZ	[\]^**

```
(] ; |. ; |. "1 ; |.!. '*' "1 ; (2: |. ])) y
```

ABCDEF	YZ[\]^	FEDCBA	*ABCDE	MNOPQR
GHIJKL	STUVWX	LKJIHG	*GHIJK	STUVWX
MNOPQR	MNOPQR	RQPONM	*MNOPQ	YZ[\]^
STUVWX	GHIJKL	XWVUTS	*STUVW	ABCDEF
YZ[\]^	ABCDEF	^]\[ZY	*YZ[\]	GHIJKL

```
1 _2 |. !. '*' 3{. y
**GHIJ
**MNOP
*****
```

Transpose

| : \_ 1 \_

Transpose

<div>  : reverses the order of the axes of its argument. For example:        ( ) ;   : ) i. 3 4 +-----+   0 1 2 3   0 4 8     4 5 6 7   1 5 9     8 9 10 11   2 6 10                3 7 11   +-----+  </div>	<div>x  : y moves axes x to the tail end. If x is boxed, the axes in each box are <i>run together</i> to produce a single axis:        y=: 3 4\$'abcdefghijkl'       y;(1 0  : y);(0  : y);((&lt;0 1)  : y) +-----+   abcd   aei   aei   afk     efgh   bfj   bfj            ijkl   cgk   cgk                   dhl   dhl          +-----+</div>
--	--

For example:

y=: a.{~ (a. i. 'a') + i. 2 3 4  
z=: y;(2 1 | : y);((<2 1) | : y);(| : i. 4 5)  
z ,&< | :&.> z

abcd	aei	afk	0 5 10 15	am	am	am	0 1 2 3 4
efgh	bfj	mrw	1 6 11 16	eq	bn	fr	5 6 7 8 9
ijkl	cgk		2 7 12 17	iu	co	kw	10 11 12 13 14
	dhl		3 8 13 18		dp		15 16 17 18 19
mnop			4 9 14 19	bn			
qrst	mqu			fr	eq		
uvwx	nrw			jv	fr		
	osw				gs		
	ptx			co	ht		
				gs			
				kw	iu		
					jv		
				dp	kw		
				ht	lx		
				lx			



## Determinant   $u \ . \ v$   2   $\_ \_$   Dot Prod

<p>The phrases <math>-/ \ . \ *</math> and <math>+/ \ . \ *</math> are the <i>determinant</i> and <i>permanent</i> of square matrix arguments. More generally, the phrase <math>u \ . \ v</math> is defined in terms of a recursive expansion by minors along the first column, as discussed below.</p>	<p>For vectors and matrices, the phrase <math>x \ +/ \ . \ * \ y</math> is equivalent to the <i>dot</i>, <i>inner</i>, or <i>matrix</i> product of math; other rank-0 verbs such as <math>&lt;.</math> and <math>*.</math> are treated analogously. In general, <math>u \ . \ v</math> is defined by <math>u@(\vee"(1+1\vee, \_))</math>, restated in English below.</p>
---	--

For example:

```

x=: 1 2 3 [ m=: >1 6 4;4 1 0;6 6 8
det=: -/ . * [ . mp=: +/ . *
x ([ ; ] ; det@] ; mp ; mp~ ; mp~@]) m
+-----+-----+-----+-----+-----+
| 1 2 3 | 1 6 4 | _112 | 27 26 28 | 25 6 42 | 49 36 36 |
|       | 4 1 0 |      |          |          | 8 25 16 |
|       | 6 6 8 |      |          |          | 78 90 88 |
+-----+-----+-----+-----+-----+

```

The monad  $u \ . \ v$  is defined as illustrated below:

```

DET=: 2 : 'v./@,`({."1 u. . v. $:@minors)@.(0:<{:@$) @ ,. "2'
minors=: }. "1 @ (1&([\..))
-/ DET * m
_112
-/ DET * 1 16 64
49
-/ DET * i.3 0
1
+/ DET * m
320

```

The definition  $u@(\vee"(1+1\vee, \_))$  given above for the dyadic case may be re-stated in words as follows:  $u$  is applied to the result of  $\vee$  on lists of “left argument cells” and the right argument *in toto*. The number of items in a list of left argument cells must agree with the number in the right argument. Thus, if  $\vee$  has ranks  $2 \ 3$  and the shapes of  $x$  and  $y$  are  $2 \ 3 \ 4 \ 5 \ 6$  and  $4 \ 7 \ 8 \ 9 \ 10 \ 11$ , then there are  $2 \ 3$  lists of left argument cells (each shaped  $4 \ 5 \ 6$ ); and if the shape of a result cell is  $sr$ , the overall shape is  $2 \ 3, sr$ .

Even , Odd     $u \ . \ . \ v$      $u \ . \ : \ v$

$$u \ . \ . \ v \leftrightarrow (u + u&v) \% 2:$$
$$u \ . \ : \ v \leftrightarrow (u - u&v) \% 2:$$

In the most commonly used case,  $v$  is arithmetic negation, and  $f=: u \ . \ . \ v$  is therefore  $f=: (u + u&-) \% 2:$ ; that is, one-half the sum of  $u \ y$  and  $u \ -y$ . The resulting function is therefore *even* in the sense that  $f \ y \leftrightarrow f \ -y$  for any  $y$ ; its graph is reflected in the vertical axis. Similarly,  $u \ . \ : \ -$  is *odd* ( $f \ y \leftrightarrow -f \ -y$ ), and its graph is reflected in the origin. Less commonly,  $v$  is matrix transpose ( $| \ :)$ , and may be any monadic function.

```
y=: _2 _1 0 1 2
1 2 3 4 5 & p. y
57 3 1 15 129
```

Polynomial with odd and even terms

```
1 2 3 4 5 & p. . . - y
93 9 1 9 93
```

Even part of polynomial

```
1 0 3 0 5 & p. y
93 9 1 9 93
```

Polynomial with even terms only

```
E=: . . -
O=: . : -
d=: 5j2&" :@ , . & . >
```

Even adverb  
Odd adverb  
Display as columns with two digits.

```
d (5&o. ; ^O ; 6&o. ; ^E ; ^ ; (^E + ^O) ; 2&o. ; ^@j.E) y
```

3.63	3.63	3.76	3.76	0.14	0.14	0.42	0.42
1.18	1.18	1.54	1.54	0.37	0.37	0.54	0.54
0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
1.18	1.18	1.54	1.54	2.72	2.72	0.54	0.54
3.63	3.63	3.76	3.76	7.39	7.39	0.42	0.42

```
m=: ?. 4 4 $ 9
(] ; (] .. | :) ; (] .: | :)) m
```

1 6 4 4	1 3.5 6 2	0 2.5 2 2
1 0 6 6	3.5 0 4.5 3	2.5 0 1.5 3
8 3 4 7	6 4.5 4 5.5	2 1.5 0 1.5
0 0 4 6	2 3 5.5 6	2 3 1.5 0

## Explicit Definition     m : n     \_ \_ \_

As defined and illustrated in Section II H, the phrase `s=: 0 : 0` may be used to define `s` as a script, and the explicit definition conjunction can be further used to produce a dyadic-only verb (`4 : s`), verb (`3 : s`), conjunction (`2 : s`), adverb (`1 : s`), or noun (`0 : s`). The left arguments `14` to `10` may be used instead of `4` to `0` to produce equivalent results in tacit form if possible. The right argument `0` may be used in each case to make the corresponding definitions directly from the keyboard: `k : 0` is equivalent to `k : (0 : 0)`. Furthermore, the boxed representation `b=: < ; _2 s` or the table representation `t=: > b` (or `t=: [ ; _2 s`) may be used in lieu of the script `s` in every case. Thus:

```
f=: 3 : 0
a=: 2+b=. y. ^ 2
a+a*b
:
x.*x.+y.
)
```

```
a=: b=: 19
f 3
110
```

```
a,b                                Only the globally assigned name is changed.
11 19
```

As illustrated by the foregoing:

1. The definitions of the monadic and dyadic cases produced by `3 : 0` are separated by a colon on a line by itself; if none occurs, the domain of the dyadic case is empty.
2. The explicit result is the result of the last non-test block sentence executed. See *Control Structures* for the definition of a test block.
3. A name assigned by the copula `=.` is made *local*; values assigned to it have no effect on the use of the same name *without* the entity defined or *within* other entities invoked by it. A name assigned by `=:` is global.
4. A locative cannot be localized.
5. The names `x.` and `y.` denote the left and right arguments. In defining a conjunction it may be necessary to refer to *its* left and right arguments (using `u.` and `v.`) as well as to the arguments of the resulting function (`x.` and `y.`). The use of `m.` instead of `u.` restricts the corresponding argument to being a noun, as does the use of `n.` instead of `v.`. For example:

**Continued**

## Explicit Definition $m : n \quad \_ \_ \_$

Continued
-----------

```

conj=: 2 : '(u. y.)+ (v. y.)'
mc=: 2 : 0
(u.y.)+(v.y.)
)

dc=: 2 : 0
:
(u.y.)+(v.x.)
)

(!conj% 2 4 5);(!mc% 2 4 5);(1 2 3 !dc% 2 4 5)
+-----+-----+-----+
| 2.5 24.25 120.2 | 2.5 24.25 120.2 | 3 24.5 120.333 |
+-----+-----+-----+

```

**Control Structures.** The sequence of execution of an explicit definition may be determined by *control words* such as `if.` `do.` `else.` `end.` and `while.`. For example, a function to find the root of a function `f` from a two-element list that brackets the root may be written and executed as follows:

```

root=: 3 : 0
m=. +/%#while. ~:/y. do. if. ~:/*f b=. (m, {.) y. do. y.=. b else. y.=. (m, {:) y. end. end. m y.
)

f=: 4:-%:
b=: 1 32
root b
16

```

Such a definition may also be written on successive lines by breaking it before or after any control word, and the foregoing definition may be made more readable as follows:

```

root=: 3 : 0
m=. +/%#
while. ~:/y.
do. if. ~:/*f b=. (m, {.) y.
do. y.=. b
else. y.=. (m, {:) y.
end.
end. m y.
)

```

**Continued**

## Explicit Definition $m : n$ \_ \_ \_

### Continued

As illustrated by the foregoing, the word `if.` and a matching `end.` mark the beginning and end of a *control structure*, as do `while.` and a matching `end.`; such structures may be *nested* as is the `if.` structure within the `while.` structure.

The control words `for.`, `if.`, `select.`, `try.`, `while.`, and `whilst.` mark the beginnings of control structures that are each terminated by a matching `end.`, and therefore provide a form of punctuation. In the foregoing example, `do.` and `else.` break the `if.` structure into three simple blocks, each comprising a sentence, whereas the `do.` in the `while.` structure breaks it into two blocks, the first being a simple sentence, and the second being itself an `if.` control structure.

In general, a *block* comprises zero or more simple sentences and control structures. The role of blocks is summarized as follows:

```
for.      T do. B end.
for_xyz.  T do. B end.
if. T do. B end.
if. T do. B else. B1 end.
if. T do. B elseif. T1 do. B1 elseif. T2 do. B2 end.
select. T case. T0 do. B0 fcase. T1 do. B1 case. T2 do. B2 end.
try. B catch. B1 end.
while. T do. B end.
whilst. T do. B end.      Like while., but Skips Test first time.
```

Words beginning with **B** or **T** denote blocks. The last sentence executed in a **T** block is tested for a non-zero value in its leading atom, and the result of the test determines the block to be executed next. (An empty **T** block result or an omitted **T** block tests true.) If an error occurs in a `try.` block, execution continues in the matching `catch.` block. The final result is the result of the last sentence executed that was not in a **T** block, and if there is no such last executed sentence, the final result is `i.0 0`.

The behaviour of the remaining control words may be summarized as follows:

```
break.      Go to end of for., while., or whilst. control structure
continue.   Go to top of for., while., or whilst. control structure
goto_name.  Go to label of same name
label_name. Target of goto_name.
return.     Exit the function
```

Additional explanations and examples can be found in the Control Structures section.

**Continued**



## Explicit Definition $m : n$ \_ \_ \_

### Continued

The following example uses control words as well as `u.` and `n.` in modelling the Level conjunction `L:` :

```

Level=: 2 : 0
m=. 0{ 3&$&|. n.
ly=. L. y. if. 0>m do. m=.0>.m+ly end.
if. m>:ly do. u. y. else. u. Level m&.> y. end.
:
'1 r'=. 1 2{ 3&$&|. n.
lx=. L. x. if. 0>l do. l=.0>. l + lx end.
ly L. x. if. 0>r do. r=.0>. r + ly end.
b=. (l,r)>:lx,ly
if. b=: 0 0 do. x. u. Level(l,r)&.>y.
elseif. b=: 0 1 do. x. u. Level(l,r)&.>y.
elseif. b=: 1 0 do. (<x.) u. Level(l,r)&.>y.
elseif. 1 do. x. u. y.
end.
)

```

```
] a=: (i.2 3);(<<2 3 4) ; 3
```

```

+-----+
| 0 1 2 | +-----+ | 3 | | |
| 3 4 5 | | +-----+ |
|       | | | 2 3 4 | |
|       | | +-----+ |
|       | | +-----+ |
+-----+

```

```
+: Level 0 a
```

```

+-----+
| 0 2 4 | +-----+ | 6 | | |
| 6 8 10 | | +-----+ |
|       | | | 4 6 8 | |
|       | | +-----+ |
|       | | +-----+ |
+-----+

```

```
+: L: 0 a
```

```

+-----+
| 0 2 4 | +-----+ | 6 | | |
| 6 8 10 | | +-----+ |
|       | | | 4 6 8 | |
|       | | +-----+ |
|       | | +-----+ |
+-----+

```

**Monad / Dyad**       $u : v \quad \_ \_ \_$

The first argument specifies the monadic case and the second argument the dyadic case.

For example:

```
y=: 10 64 100
^._ y
2.30259 4.15888 4.60517
Natural logarithm

10&^._ y
1 1.80618 2
Base ten logarithm

log=: 10&^._ : ^._
log y
1 1.80618 2

8 log y
1.10731 2 2.21462

(^1) log y
2.30259 4.15888 4.60517

LOG=: 10&$ : ^._
LOG y
1 1.80618 2
Use of self-reference

f=: % : ($:@-)
(f y),: 100 f y
3.16228 8 10
9.48683 6 0

ABS=: | : [:
ABS _4
4

3 ABS _4
|valence error: ABS
| 3 ABS _4
```

The domain of the dyad `ABS` is empty because the domain of `| :` is empty.

# Obverse      $u :. v$      $mu$ $lu$ $ru$

The result of  $u :. v$  is the verb  $u$ , but with an assigned obverse  $v$  (used as the “inverse” under the conjunctions  $\&.$  and  $\wedge:$ ).

For example:

```

y=: _4 0 4 3j4
rp=: <@(%: , -@%:)"0
rp y
-----+
|0j2 0j_2|0 0|2 _2|2j1 _2j_1|
-----+
I=: ^: _1
rp I
rp^:_1
rp I rp y
|domain error
|      rp I rp y
inv=: *:@{. @, @>
inv rp y
_4 0 4 3j4

RP=: rp :. inv
RP I RP y
_4 0 4 3j4

rc=: <@(: +)@(: -)@%:"0
rc y
-----+
| 0j2 0j_2|0 0|2 _2| 2j1 _2j_1|
|0j_2 0j2|0 0|2 _2|2j_1 _2j1|
-----+
RC=: rc :. inv
RC I RC y
_4 0 4 3j4

```

Root pairs

No assigned obverse

Assigned obverse in RP

Root companions

## Adverse $u :: v$ — — —

The result of  $u :: v$  is that of  $u$ , provided that  $u$  completes without error; otherwise the result is the result of  $v$ .

For example:

<pre> p=: 3 1 0 2 x=: 'ABCD' p{x DBAC </pre>	<p>A permutation vector</p>
<pre>  i=: A. p 20 </pre>	<p>Atomic index in ordered list of permutations</p>
<pre> i A. x DBAC </pre>	<p>Permutation by atomic representation</p>
<pre> q=: 3 1 1 0 q{x DBBA </pre>	<p>Not a permutation</p>
<pre> A. q  index error   A.q </pre>	
<pre> A=: A. :: (!@#) A p 20 </pre>	<p>Give index outside range in case of error</p>
<pre> A q 24 </pre>	
<pre> 24 A. x  index error   24 A.x </pre>	

# Ravel

, \_ \_ \_

# Append

<p><code>,y</code> gives a list of the atoms of <code>y</code> in “normal” order: the result is ordered by items, by items within items, etc. The result shape is <code>1\$*/\$ y</code>. Thus:</p> <pre> y=: 2 4 \$ 'abcdefgh'  y abcd efgh  ,y abcdefgh </pre>	<p><code>x,y</code> appends items of <code>y</code> to items of <code>x</code> after:</p> <ol style="list-style-type: none"> <li>1) Reshaping an atomic argument to the shape of the items of the other,</li> <li>2) Bringing the arguments to a common rank (of at least 1) by repeatedly <i>itemizing</i> (<code>,:</code>) any of lower rank, and</li> <li>3) Bringing them to a common shape by padding with fill elements in the manner described in Section II B.</li> </ol> <p>The fit conjunction (<code>,!f</code>) provides fill specified by the items of <code>f</code>.</p>
--	--

```

]a=: i. 2 3 3
0 1 2
3 4 5
6 7 8

9 10 11
12 13 14
15 16 17

,a
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

,"2 a
0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17

```

The following examples illustrate the dyadic case:

```

('abc', 'de'); ('abc', "0/'de'); (5 6 7, i.2 3); (7, i.2 3)
+-----+
|abcde|ad|5 6 7|7 7 7|
|      |ae|0 1 2|0 1 2|
|      |  |3 4 5|3 4 5|
|      |bd|      |      |
|      |be|      |      |
|      |cd|      |      |
|      |ce|      |      |
+-----+

```

Ravel Items

, . \_ \_ \_

Stitch

<p>If <i>y</i> is an atom, then <i>,.y</i> is <i>1 1\$y</i>; otherwise, <i>,.y</i> is <i>, "_1 y</i>, the table formed by ravelling each item of <i>y</i>.</p>	<p><i>x,.y</i> is equivalent to <i>x, "_1 y</i>. In other words, items of <i>x</i> are stitched to corresponding items of <i>y</i>.</p> <p>The fit conjunction (<i>,.!.f</i>) provides fill specified by the items of <i>f</i>.</p>
--	---

For example:

```
a=: i. 2 3 2
($,.3) ; (,.2 3 5 7 11) ; ($,.<'abcd') ; a ; (,.a)
```

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

The following examples illustrate the dyadic case:

```
b=:3 4$'abcdefghijl' [ c=:3 4$'ABCDEFGHijkl'
b ; c ; (b,.c) ; (b,c) ; a ; (a ,. |."1 a) ; (,/a) ; (,./a)
```

abcd	ABCD	abcdABCD	abcd	0 1	0 1	0 1	0 1 6 7
efgh	EFGH	efghEFGH	efgh	2 3	2 3	2 3	2 3 8 9
ijkl	IJKL	ijklIJKL	ijkl	4 5	4 5	4 5	4 5 10 11
			ABCD		1 0	6 7	
			EFGH	6 7	3 2	8 9	
			IJKL	8 9	5 4	10 11	
				10 11			
					6 7		
					8 9		
					10 11		
					7 6		
					9 8		
					11 10		

# Itemize

, :

— — —

# Laminate

,:y adds a leading unit axis to y, giving a result of shape 1,\$y. Thus:

```
$ ,: 2 3 4
1 3
```

An atomic argument in x,:y is first reshaped to the shape of the other (or to a list if the other argument is also atomic); the results are then itemized and catenated, as in (,:x),(,:y).

The fit conjunction (,:!.f) provides fill specified by the items of f.

```
s=: 3 [ v=: 2 3 4 [ m=: i. 3 3
(,:s); ($,: s); (:v); ($,:v); ($,:m); ($,:^:4 v)
+---+
|3|1|2 3 4|1 3|1 3 3|1 1 1 1 3|
+---+
```

The following examples compare the dyadic cases of Append and Laminate:

```
a=: 'abcd' [ A=: 'ABCD' [ b=: 'abcdef'
(a,A) ; (a,:A) ; (a,:b) ; (m,m) ; (m ,: m)
```

```
+---+
|abcdABCD|abcd|abcd|0 1 2|0 1 2|
|          |ABCD|abcdef|3 4 5|3 4 5|
|          |      |      |6 7 8|6 7 8|
|          |      |      |0 1 2|0 1 2|
|          |      |      |3 4 5|3 4 5|
|          |      |      |6 7 8|6 7 8|
+---+
```

```
t=: i. 3 2 2
t ; (:/t) ; (,./t) ; (:/t)
+---+
|0 1|0 1|0 1 4 5 8 9|0 1|
|2 3|2 3|2 3 6 7 10 11|2 3|
|   |4 5|   |   |
|4 5|6 7|   |   |
|6 7|8 9|   |   |
|   |10 11|   |   |
|8 9|   |   |   |
|10 11|   |   |   |
|   |   |   |   |
|   |   |   |   |
|   |   |   |   |
|   |   |   |   |
+---+
```

Raze

i

—

—

—

Link

<i>y</i> assembles along a leading axis the opened elements of the ravel of <i>y</i> .	<i>x</i> <i>y</i> is (< <i>x</i> ), <i>y</i> if <i>y</i> is boxed, and (< <i>x</i> ),< <i>y</i> if <i>y</i> is open.
--	--

```
]bv=: 1 2 3;4 5 6;7 8 9
+-----+
|1 2 3|4 5 6|7 8 9|
+-----+

;bv
1 2 3 4 5 6 7 8 9

]m=: >bv
1 2 3
4 5 6
7 8 9

;/ m
+-----+
|1 2 3|4 5 6|7 8 9|
+-----+

(i;/1 2 3 4 5) ,&< (i/i. 3 4)
+-----+
|+-----+|+-----+| | | | | | | |
|1|2|3|4|5| |0 1 2 3|4 5 6 7|8 9 10 11|
|+-----+|+-----+|
+-----+

]txt=: '3 %: 4 ^. 5'
3 %: 4 ^. 5

]s=: ;: txt           Word formation
+-----+
|3|%:|4|^.|5|
+-----+

;s
3%:4^.5

(boxifopen=: <^:(< -: {:@i~)) 3 4
+----+
|3 4|
+----+

(<3 4) = boxifopen <3 4
1
```



# Cut      $m; .n$      $u; .n$      $\_1/2$      Cut

<p><math>u; .0</math> <math>y</math> applies <math>u</math> to <math>y</math> after reversing <math>y</math> along each axis; it is equivalent to <math>(0 \_1 * / \\$y) u; .0 y</math>.</p> <p>The <i>fret</i> <math>0\{y</math> (the leading item of <math>y</math>) marks the start of an interval of items of <math>y</math>; the phrase <math>u; .1 y</math> applies <math>u</math> to each such interval. The phrase <math>u; .\_1 y</math> differs only in that frets are excluded from the result. In <math>u; .2</math> and <math>u; .\_2</math> the fret is the <i>last</i> item, and marks the <i>ends</i> of intervals.</p> <p>The monads <math>u; .3</math> and <math>u; .\_3</math> apply <math>u</math> to tessellation by “maximal cubes”, that is, they are defined by their dyadic cases using the left argument <math>(\\$ \\$y) \\$&lt; . / \\$y</math>.</p> <p><math>m; .n y</math> applies successive verbs from the gerund <math>m</math> to the cuts of <math>y</math>, extending <math>m</math> cyclically as required.</p>	<p><math>x u; .0 y</math> applies <math>u</math> to a rectangle or cuboid of <math>y</math> with one vertex at the point in <math>y</math> indexed by <math>v = :0\{x</math>, and with the opposite vertex determined as follows: the dimension is <math> 1\{x</math>, but the rectangle extends <i>back</i> from <math>v</math> along any axis <math>j</math> for which the index <math>j\{v</math> is negative. Finally, the order of the selected items is reversed along each axis <math>k</math> for which <math>k\{1\{x</math> is negative. If <math>x</math> is a vector, it is treated as the matrix <math>0, :x</math>.</p> <p>The frets in the dyadic cases <math>\_1</math>, <math>\_1</math>, <math>\_2</math>, and <math>\_2</math> are determined by the <math>\_1</math> in the boolean vector <math>x</math>.</p> <p><math>u; .3</math> and <math>u; .\_3</math> yield (possibly overlapping) tessellations. <math>x u; .\_3 y</math> applies <math>u</math> to each <i>complete</i> rectangle of size <math> 1\{x</math> beginning at integer multiples of (each item of) the movement vector <math>0\{x</math>. As in <math>u; .0</math>, reversal occurs along each axis for which the size <math>1\{x</math> is negative. The case of a list <math>x</math> is equivalent to <math>\_1, :x</math>, and therefore provides a complete tessellation of size <math>x</math>. The case <math>u; .3</math> differs in that shards of length less than <math> 1\{x</math> are included.</p> <p><math>x m; .n y</math> applies successive verbs from the gerund <math>m</math> to the cuts of <math>y</math>, extending <math>m</math> cyclically as required.</p> <p>The 0- and 3-cuts have a left rank of 2; the 1- and 2-cuts have a left rank of 1.</p>
--	--

Continued

Cut m; .n u; .n \_ 1/2 \_ Cut

Continued

y=: 'worlds on worlds '  
(<i.2 y) i (\$i.\_2 y) i (3 5\$i.10) i (+/ i.1 (3 5\$i.10))

worlds	on	worlds	6	0	1	2	3	4	5	7	9	11	13
			2	5	6	7	8	9	0	1	2	3	4
			6	0	1	2	3	4					

x=:1 \_2,:\_2 3 [ z=: i. 5 5  
x i (x ]i.0 z) i z

1 _2	11 12 13	0	1	2	3	4
_2 3	6 7 8	5	6	7	8	9
		10	11	12	13	14
		15	16	17	18	19
		20	21	22	23	24

(y=: a. {~ (a. i. 'a') + i. 4 4);(a=: 1 1 ,: 2 2)

abcd	1 1
efgh	2 2
ijkl	
mnop	

(<i.3 y) i (((\$\$y)\$<./\$y)<i.3 y) i (a <i.3 y) i <(a <i.\_3 y)

abcd	bcd	cd	d	abcd	bcd	cd	d	ab	bc	cd	d	ab	bc	cd
efgh	fgh	gh	h	efgh	fgh	gh	h	ef	fg	gh	h	ef	fg	gh
ijkl	jkl	kl	l	ijkl	jkl	kl	l							
mnop	nop	op	p	mnop	nop	op	p	ef	fg	gh	h	ef	fg	gh
								ij	jk	kl	l	ij	jk	kl
efgh	fgh	gh	h	efgh	fgh	gh	h							
ijkl	jkl	kl	l	ijkl	jkl	kl	l	ij	jk	kl	l	ij	jk	kl
mnop	nop	op	p	mnop	nop	op	p	mn	no	op	p	mn	no	op
ijkl	jkl	kl	l	ijkl	jkl	kl	l	mn	no	op	p			
mnop	nop	op	p	mnop	nop	op	p							
mnop	nop	op	p	mnop	nop	op	p							

# Word Formation ; : 1

;:y is the list of boxed words in the list y according to the rhematic rules of Part I. The function also applies reasonably well to ordinary text.	
--	--

```

s=: '*: @ -: @ i. 2 3'
do=: ".
do s
  0 0.25      1
2.25      4 6.25

;: s
+---+---+---+---+
|*:|@| -:|@|i.|2 3|
+---+---+---+---+

; ;: s
*:@-:@i.2 3

p=: 'When eras die, their legacies/'
q=: 'are left to strange police'
r=: 'Professors in New England guard'
s=: 'the glory that was Greece'
;: p
+---+---+---+---+
|When|eras|die|,|their|legacies|/|
+---+---+---+---+

> ;: p,q
When
eras
die
,
their
legacies
/
are
left
to
strange
police

|. &. ;: p
/ legacies their , die eras When

```

Tally

#

1

Copy

<div>#y is the number of items in y. Thus: <div>(# ' '); (# 'a '); (# 'ab ')</div><div>+--+--+  0 1 2  +--+--+  (#3); (#,3); (# 3 4)</div><div>+--+--+  1 1 2  +--+--+  (#i.4 5 6); (#\$i.4 5 6)</div><div>+--+--+  4 3  +--+--+</div></div>	<div>If the arguments have an equal number of items, then x#y copies +/x items from y, with i{x repetitions of item i{y. Otherwise, if one is an atom it is repeated to make the item count of the arguments equal.  The complex left argument a j. b copies a items followed by b fills. The fit conjunction provides specified fills, as in #!.f.</div>
--	---

Copy is illustrated by the following examples:

0 1 2 3 4 5 # 0 1 2 3 4 5  
1 2 2 3 3 3 4 4 4 4 5 5 5 5 5  
  
t=: 3 4 \$'abcdefghijkl' [ n=: i. 3 4  
  
t ; n ; (3 0 1 # t) ; (3 0 1 # n) ; (3 1 4 2 #"1 t)  

+-----+  
abcd	0 1 2 3	abcd	0 1 2 3	aaabccccdd
efgh	4 5 6 7	abcd	0 1 2 3	eeefgggghh
ijkl	8 9 10 11	abcd	0 1 2 3	iiijkkkkll
ijkl	8 9 10 11			
+-----+

  
k=: 2j1 0 1j2  
(k # t);(k # n);(k #!.'\*' t);(k #!.4 n)  

+-----+  
abcd	0 1 2 3	abcd	0 1 2 3
abcd	0 1 2 3	abcd	0 1 2 3
	0 0 0 0	\*\*\*\*	4 4 4 4
ijkl	8 9 10 11	ijkl	8 9 10 11
	0 0 0 0	\*\*\*\*	4 4 4 4
	0 0 0 0	\*\*\*\*	4 4 4 4
+-----+

**Base Two**

# . 1 1 1

**Base**

#.y is the base-2 value of y, that is,  
2#.y. For example:

```
    #. 1 0 1 0
10
```

```
    #. 2 3$ 0 0 1,1 0 1
1 5
```

x#.y is a weighted sum of the items of y;  
that is,  $\sum w_i y_i$ , where w is the product  
scan  $\ast/\backslash.\}$ .x,1. An atomic argument is  
reshaped to the shape of the other argument.

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

```
    10 #.a
123 4567 9011
```

```
    8 #. a
83 2423 4763
```

```
    ]time=: 0 1 3,1 1 3,:2 4 6
0 1 3
1 1 3
2 4 6
```

```
    x=: 24 60 60
    x #. time
63 3663 7446
```

```
    x,1
24 60 60 1
```

```
    ]w=:  $\ast/\backslash.\}$ . x,1
3600 60 1
```

```
    w *"1 time
0 60 3
3600 60 3
7200 240 6
```

**Continued**

Base Two

# . 1 1 1

Base

Continued

```
+/"1 w *"1 time
63 3663 7446

w +/@"* "1 time
63 3663 7446

c=: 3 1 4 2 [ y=: 0 1 2 3 4 5
c p. y          Polynomial with coefficients c
3 10 37 96 199 358

y #."0 1 |.c
3 10 37 96 199 358
```

## Antibase Two      # :         1 0      Antibase

#:  $y$  is the binary representation of  $y$ , and is equivalent to  $(m\#2)\#y$ , where  $m$  is the maximum of the number of digits needed to represent the atoms of  $y$  in base 2. For example:

```

      i. 8
0 1 2 3 4 5 6 7
      #: i. 8
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

```

In simple cases  $r\&\#:$  is inverse to  $r\&\#.$ . Thus:

```

      r=: 24 60 60
      r #: r #. 2 3 4
2 3 4

```

But if  $r \#. y$  exceeds  $(*/r)-1$  (the largest integer representable in the radix  $r$ ), then the result of  $r\#y$  is reduced modulo  $*/r$ . For example:

```

      r #: r #. 29 3 4
5 3 4

```

A representation in an arbitrary base that is analogous to the base-2 representation provided by the monadic use of #: may be provided as illustrated below:

```

      ndr=: 1: + <.@^
      10 ndr y=: 9 10 11 100 99 100
1 2 2 3 2 3

```

```

      (y#::~~10 #~ >./10 ndr y);(y#::~~8 #~ >./8 ndr y)

```

```

+-----+
| 0 0 9 | 0 1 1 |
| 0 1 0 | 0 1 2 |
| 0 1 1 | 0 1 3 |
| 1 0 0 | 1 4 4 |
| 0 9 9 | 1 4 3 |
| 1 0 0 | 1 4 4 |
+-----+

```

```

      (10&#. ^:_1 ; 8&#. ^:_1) y

```

```

+-----+
| 0 0 9 | 0 1 1 |
| 0 1 0 | 0 1 2 |
| 0 1 1 | 0 1 3 |
| 1 0 0 | 1 4 4 |
| 0 9 9 | 1 4 3 |
| 1 0 0 | 1 4 4 |
+-----+

```

## Factorial ! 0 0 0 Out Of (Combinations)

<p>For a non-negative integer argument <math>y</math>, the definition is <math>x/y \equiv i.y</math>. In general, <math>i.y</math> is <math>\Gamma(1+y)</math> (the gamma function). Thus:</p> <pre> (* / 1 2 3 4 5) , (! 5) 120 120  ]x=: 2 %~ 3 ~ i. 2 4 _1.5 _1 _0.5 0 0.5 1 1.5 2  !x _3.54491 _ 1.77245 1 0.886227 1 1.32934 2  ]fi=: !^:_1(24 25 2.1 9876) 4 4.02705 2.05229 7.33019  ! fi 24 25 2.1 9876 </pre>	<p>For non-negative arguments <math>x!y</math> is the number of ways that <math>x</math> things can be chosen out of <math>y</math>. More generally, <math>x!y</math> is <math>(!y) \% (!x) * (!y-x)</math>. Thus:</p> <pre> 3!5 10 (!5) \% (!3) * (!5-3) 10 1j2 ! 3.5 8.64269j16.9189  ]y=: 2&amp;!^:_1 (45 4.1 30 123) 10 3.40689 8.26209 16.1924 2 ! y 45 4.1 30 123  ]x=: !&amp;10^:_1 (2.5 45) 0.3433618 2 x ! 10 2.5 45 </pre>
--	--

The first table below illustrates the relation between the dyad  $!$  and the table of binomial coefficients; the last two illustrate its relation to the *figurate numbers*:

```

h=: 0,i=: i.5 [ j=: -1+i.5 [ k=: 5#1
tables=: ( ,.h);(i,i!/i);(j,i!/j);(k,i(+/\^:)k)
format=: ({. ,:&< }. )@" :&. >
format tables

```

0	0	1	2	3	4	_1	_2	_3	_4	_5	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	2	3	4	_1	_2	_3	_4	_5	1	2	3	4	5
2	0	0	1	3	6	1	3	6	10	15	1	3	6	10	15
3	0	0	0	1	4	_1	_4	_10	_20	_35	1	4	10	20	35
4	0	0	0	0	1	1	5	15	35	70	1	5	15	35	70

Figurate numbers of order zero are all ones; those of higher orders result from successive applications of subtotals (that is, sums over prefixes, or  $+/\backslash$ ). Those of order two are the *triangular numbers*, resulting from subtotals over the integers beginning with one.



## Fit (Customize) ! .

This conjunction modifies certain verbs in ways prescribed in their definitions. For example, `=! .t` is the relation of equality using tolerance `t`, and `^! .r` is the *factorial function* so defined that `x ^! .r n` is `*/x + r * i. n`. Consequently, `^! .__1` is the *falling factorial function*.

Fit applies to the following verbs (to produce *variants*). The monadic case is shown before a bullet, and the dyadic case after it:

<code>&lt; &lt;: &gt; &gt;: +. *. -. -:   E. i. i:</code>	• Tolerance
<code>&lt;. &gt;. * ~.</code>	Tolerance •
<code>#:</code>	• Tolerance
<code>= ~: #: e.</code>	Tolerance • Tolerance
<code>^ p.</code>	• Stope function and polynomial based thereon
<code>\$  . , ,. ,: # {.</code>	• Fill
<code>" :</code>	Print precision •

# Foreign ! :

This conjunction is used to communicate with the host system as well as with the keyboard (as an input file) and with the screen (as an output file). It is also used to provide a variety of extra-lingual facilities, such as setting the form of function display, determining the class of a name (noun, verb, adverb, or conjunction), and listing all existing names in specified classes.

```
(mean=: +/ % #) a=: 2 3 5 7 11 13
6.83333
```

```
mean
+/ % #
```

```
9!:3 (4)
mean
+- / --- +
--+- %
+- #
```

Tree display of verb

```
9!:3 (2 4 5)
mean
+-----+-----+
|+---+| % | # |
|+---+| | |
|+---+| | |
+-----+-----+
```

Boxed display

```
+- / --- +
--+- %
+- #
```

Tree display

```
+/ % #
```

Linear display

```
4!:0 'a' ; 'mean'
0 3
```

Classes of names (noun 0, verb 3)

```
4!:1 (3)
+-----+
|mean|
+-----+
```

List of names in class 3

The appendix shows all uses of the foreign conjunction.

The *identity function* of  $u$  is a function  $\text{id}_u$  such that  $\text{id}_u y$  is  $u/y$  if  $0 \neq y$ . The identity functions used are:

**For**

```
< > + - +. ~: | (2 4 5 6 b.)
= <: >: * % *. %: ^ ! (1 9 11 13

<.
>.

'
C. {
%. +/ . *
u/
u&.v
```

Oblique

m/.      u/.      \_ \_ \_

Key

<p>u/.y applies u to each of the oblique lines of a table y. For example:</p> <div><div>i.3 4</div><div>0 1 2 3</div><div>4 5 6 7</div><div>8 9 10 11</div></div> <div><div>&lt;/. i.3 4</div><div>+-----+</div><div> 0 1 4 2 5 8 3 6 9 7 10 11 </div><div>+-----+</div></div> <p>In general, u/.y is the result of applying u to the oblique lines of _2-cells of y. If the rank of y is less than two, y is treated as the table ,.y.</p> <p>m/.y applies successive verbs from the gerund m to the oblique lines of _2-cells of y, extending m cyclically as required.</p> <p>Thus:</p> <div><div>&lt;`(&lt;@ ..) /. i.3 4</div><div>+-----+</div><div> 0 4 1 2 5 8 9 6 3 7 10 11 </div><div>+-----+</div></div>	<p>x u/.y ↔ (=x) u@# y, that is, items of x specify keys for corresponding items of y and u is applied to each collection of y having identical keys. For example:</p> <div><div>1 2 3 1 3 2 1 &lt;/. 'abcdefg'</div><div>+-----+</div><div> adg bf ce </div><div>+-----+</div></div> <p>x m/.y applies successive verbs from the gerund m to the collections of y, extending m cyclically as required.</p>
---	---

Continued

# Oblique      m/ .      u/ .      \_ \_ \_      Key

## Continued

The application of a function to diagonals of a table is commonly useful, as in correlation, in convolution, and in products of polynomial coefficients (or, equivalently, products of numbers in a fixed base). For example:

```
t=: p */ q [ p=: 1 2 1 [ q=: 1 3 3 1
t ; (+//.t) ; 1 1 &(+//.@(*//)) ^: (i.6) 1
```

1 3 3 1	1 5 10 10 5 1	1 0 0 0 0 0
2 6 6 2		1 1 0 0 0 0
1 3 3 1		1 2 1 0 0 0
		1 3 3 1 0 0
		1 4 6 4 1 0
		1 5 10 10 5 1

```
((10#.p)*10#.q), 10 #. +//. p */ q
161051 161051
```

Unlike polynomial coefficients, the diagonal sums of a multiplication table of digits should be “normalized” if any equal or exceed the radix.

Grade Up

/ : \_ \_ \_

Sort Up

<pre>/: grades any argument, yielding a permutation vector; (/:y){y sorts y in ascending order. For example:  n=: 3 1 4 2 1 3 3 ]g=: /: n 1 4 3 0 5 6 2  g { n 1 1 2 3 3 3 4</pre>	<pre>x/:y is (/:y){x; i.e., x is sorted to an order specified by y. In particular, y/:y (or /:~y) sorts y. For example:  y=: 'popfly' y /: 3 1 4 1 5 9 ofpply  y /: y floppy</pre>
--	--

Elements of `/:y` that select equal elements of `y` are in ascending order. If `y` is a table, `/:y` grades the base value of the rows, using a base larger than twice the magnitude of any of the elements. Higher ranks are treated as `,.y`, (as if its items were each ravelled).

If `y` is literal, `/:y` grades according to the collating sequence determined by the alphabet `a.`; another collating sequence `cs` can be imposed by grading `cs i. y`. For example:

```
]n=: 3 1 4 1 6,2 7 1 8 3,:6 1 8 0 3
3 1 4 1 6
2 7 1 8 3
6 1 8 0 3

/: n
1 0 2
```

```
Aa=: ' ',. a. {~ 65 97 +/ i. 26
x=: words=: >: 'When eras die'
j=: <./Aa i."1 _ x
x ; (x/:x) ; (x/:j) ; Aa
```

When	When	die	ABCDEFGHIJKLMNOPQRSTUVWXYZ
eras	die	eras	abcdefghijklmnopqrstuvwxyz
die	eras	When	

The three types: numeric or empty, literal, and boxed, are so ordered; within them, a lower rank precedes a higher, and a smaller shape precedes a larger. Complex arguments are ordered by real part, then by imaginary. Boxed arrays are ordered according to the opened elements.

# Prefix $m \backslash$ $u \backslash$ $\_ 0 \_$ Infix

<p><math>u \backslash y</math> has <math>\#y</math> items resulting from applying <math>u</math> to each of the prefixes <math>k\{.y</math>, for <math>k</math> from 1 to <math>\#y</math>.</p> <p><math>m \backslash y</math> applies successive verbs from the gerund <math>m</math> to the prefixes of <math>y</math>, extending <math>m</math> cyclically as required.</p>	<p>If <math>x &gt; 0</math>, the items of <math>x \ u \backslash y</math> result from applying <math>u</math> to each infix of length <math>x</math>. If <math>x &lt; 0</math>, <math>u</math> is applied to <i>non-overlapping</i> infixes of length <math> x </math>, including any final shard.</p> <p><math>x \ m \backslash y</math> applies successive verbs from the gerund <math>m</math> to the infixes of <math>y</math>, extending <math>m</math> cyclically as required.</p>
--	--

```

+/\a=: 1 2 4 8 16
1 3 7 15 31

```

Subtotals, or partial sums

```

*/\a
1 2 8 64 1024

```

Partial products

```

<\a
+---+---+---+---+
|1|1 2|1 2 4|1 2 4 8|1 2 4 8 16|
+---+---+---+---+

```

```

<\i.3 4
+---+---+---+---+
|0 1 2 3|0 1 2 3|0 1 2 3|
|         |4 5 6 7|4 5 6 7|
|         |8 9 10 11|
+---+---+---+---+

```

```

(+/\^:_1 +/\ a) ,: */\^:_1 a
1 2 4 8 16
1 2 2 2 2

```

The following examples illustrate the use of the dyad *infix*:

```

((2: -/\ ]) ; (2: --/\ ])) a
+---+---+---+---+
|_1 _2 _4 _8|1 2 4 8|
+---+---+---+---+

```

Backward and forward differences

```

((3: <\ ]) ,&< (_3: <\ ])) 'abcdefgh'
+---+---+---+---+
|+---+---+---+---+|+---+---+---+| | | | | | | | |
|abc|bcd|cde|def|efg|fgh| |abc|def|gh|
|+---+---+---+---+|+---+---+---+|
+---+---+---+---+

```

Suffixm\ .u\ . \_ 0 \_ Outfix

<p>u\ .y has #y items resulting from applying u to suffixes of y, beginning with one of length #y (that is, y itself), and continuing through a suffix of length 1.</p> <p>m\ .y applies successive verbs from the gerund m to the suffixes of y, extending m cyclically as required.</p>	<p>If x&gt;:0 in x u\ . y, then u applies to outfixes of y obtained by suppressing successive infixes of length x. If x&lt;0, the outfixes result from suppressing non-overlapping infixes, the last of which may be a shard.</p> <p>x m\ .y applies successive verbs from the gerund m to the outfixes of y, extending m cyclically as required.</p>
---	---

```
*/\ . y=: 1 2 3 4 5
120 120 60 20 5

<\ . y
+-----+-----+-----+-----+
|1 2 3 4 5|2 3 4 5|3 4 5|4 5|5|
+-----+-----+-----+-----+

3 <\ . 'abcdefgh'
+-----+-----+-----+-----+
|defgh|aefgh|abfgh|abcgh|abcdh|abcde|
+-----+-----+-----+-----+

_3 <G\ . 'abcdefgh'
+-----+-----+
|defgh|abcgh|abcdef|
+-----+-----+

]m=: i.3 3
0 1 2
3 4 5
6 7 8

<"_2 (minors=: 1&( |:\ . )"2^:2) m
+-----+-----+
|4 5|3 5|3 4|
|7 8|6 8|6 7|
+-----+-----+
|1 2|0 2|0 1|
|7 8|6 8|6 7|
+-----+-----+
|1 2|0 2|0 1|
|4 5|3 5|3 4|
+-----+-----+
```



# Grade Down $\backslash :$ Sort Down

$\backslash :$ grades <i>any</i> argument, yielding a permutation vector; $(\backslash :y)\{y$ sorts $y$ in descending order. For example: <pre> ]g=: \:y=: 3 1 4 2 1 3 3 2 0 5 6 3 1 4  g{y 4 3 3 3 2 1 1 </pre>	$x\backslash :y$ is $(\backslash :y)\{x$ ; i.e., $x$ is sorted to an order specified by $y$ . In particular, $y\backslash :y$ (or $\backslash :~y$ ) sorts $y$ . For example: <pre> \::~"1 'dozen',: 'disk' zoned skid </pre>
--	---

Elements of  $\backslash :y$  that select equal elements of  $y$  are in ascending order. If  $y$  is a table,  $\backslash :y$  grades the base value of the rows, using a base larger than twice the magnitude of any of the elements. Higher ranks are treated as  $,.y$  (as if the items were each ravelled).

If  $y$  is literal,  $\backslash :y$  grades according to the collating sequence specified by the alphabet  $a.$ ; another collating sequence  $cs$  can be imposed by grading  $cs$  i.  $y$ . For example:

```

]n=: 3 1 4 1 6,2 7 1 8 3,:6 1 8 0 3
3 1 4 1 6
2 7 1 8 3
6 1 8 0 3

\:: n
2 0 1

\::~>:'when eras die, their legacies'
when
their
legacies
eras
die
,
```

See Grade Up ( $/:$ ) for the treatment of complex and boxed arguments.

Same

[     ]     \_   \_   \_

Left, Right

The monads [ and ] are each <i>identity</i> functions; each yields its argument.	x [ y ( <i>left bracket</i> ) yields the left argument x, and x ] y yields the right argument y.
--	--

For example:

```
n=: i. 2 3
a=: 'abcde'

]n
0 1 2
3 4 5

[a
abcde

n[a
0 1 2
3 4 5

n]a
abcde

([ \ ; ] \ ; [ \. ; ] \.) 'ABCDEF'
```

A	A	ABCDEF	ABCDEF
AB	AB	BCDEF	BCDEF
ABC	ABC	CDEF	CDEF
ABCD	ABCD	DEF	DEF
ABCDE	ABCDE	EF	EF
ABCDEF	ABCDEF	F	F

See the corresponding *conjunctions* [ . and ] . (Lev and Dex).

## Lev, Dex, Identity [ . ] . ] :

[ . is a conjunction that yields the left argument and ] . is a conjunction that yields its right argument.

] : is the identity adverb, that yields its argument.

For example:

```

      sqrt=: %: [ . sqr=: *:
      sqrt 1 2 3
1 1.41421 1.73205

      sqr 1 2 3
1 4 9

      ^ ]: 0 1 2 3
1 2.71828 7.38906 20.0855

```

**Cap****[ : \_ \_ \_****Cap**

[ : caps a left branch of a fork, as described in Section II F. For example, the function  
 p=: [ : +/ + \* - applies the monad +/ to the result of the fork + \* - .

Caps make it possible to define a wider range of functions as unbroken trains. For example, the maximum divided by the product of the sum and difference would be defined by a single train, whereas (without the use of the cap) the definition of the maximum divided by the (monad) floor of the product of the sum and difference would require the use of trains interrupted by the monad. Thus:

```
f=: >. % + * -
g=: >. % <. @ (+ * -)
2.5 f 4
_0.410256

2.5 g 4
_0.4
```

The cap makes possible the use of an unbroken train as follows:

```
h=: >. % [ : <. + * -
2.5 h 4
_0.4
```

Since the domain of the cap is empty, it can be used (with : ) to define a function whose monadic or dyadic case invokes an error. For example:

```
abs=: | : [ :
res=: [ : : |
res _4 0 5
|valence error: res
res _4 0 5
abs _4 0 5
4 0 5

3 res _4 0 5
2 0 2

3 abs _4 0 5
|valence error: abs
| 3 abs _4 0 5
```

# Catalogue { 1 0 \_ From

{y forms a catalogue from the atoms of its argument, its shape being the chain of the shapes of the opened items of y. The common shape of the boxed results is \$y. For example:

```

      { 'ht'; 'ao'; 'gtw'
+----+----+----+
| hag| hat| haw|
+----+----+----+
| hog| hot| how|
+----+----+----+
| tag| tat| taw|
+----+----+----+
| tog| tot| tow|
+----+----+----+

```

The Cartesian product is readily defined in terms of {, thus:

```

      CP=: { @ ( , & < )
      0 1 CP 2 3 4
+----+----+----+
| 0 2| 0 3| 0 4|
+----+----+----+
| 1 2| 1 3| 1 4|
+----+----+----+

```

If x is an integer in the range from -n=: #y to n-1, then x{y selects item n|x from y. Thus:

```

      2 0 _1 _3 { 'abcdefg'
cage

      t=: 3 4$'abcdefghijkl'
      1{t
efgh

```

More generally, >x may be a list whose successive elements are (possibly) boxed arrays that specify selection along successive axes of y.

Finally, if any r=:>j{>x used in the selection is itself boxed, selection is made by the indices along that axis that *do not* occur in >r.

Note that the result in the very last dyadic example, that is, (<<<\_1){m, is all *except* the last item.

```

      t; (1{t); (2 1{t); (1{"1 t); ((,1){"1 t); (2 1{"1 t)
+----+----+----+----+----+
|abcd|efgh|ijkl|bfj|b|cb|
|efgh|      |efgh|    |f|gf|
|ijkl|      |    |    |j|kj|
+----+----+----+----+----+

```

Continued

Catalogue

{ 1 0 \_

From

Continued

t; (2 0{t); ((<2 0){t); ((2 0;1 3){t); ((<2 0;1 3){t)

abcd	ijkl	i	ih	jl
efgh	abcd			bd
ijkl				

(\_1{m); (\_1{"2 m); (\_1{"1 m); (<<<\_1){m=:i.2 3 4

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

# Head { . 1 \_ Take

<p>{.y selects the leading item of y. Thus:</p> <pre>a=: i. 3 2 3 a;({.a);({. "2 a);({. "1 a)</pre> <pre>+-----+-----+-----+-----+   0 1 2   0 1 2   0 1 2   0 3     3 4 5   3 4 5   6 7 8   6 9                     12 13 14   12 15     6 7 8                             9 10 11                             12 13 14                             15 16 17                           +-----+-----+-----+-----+</pre> <pre>]b=: i/a</pre> <pre>+-----+-----+-----+   0 1 2   6 7 8   12 13 14     3 4 5   9 10 11   15 16 17   +-----+-----+-----+</pre> <pre>{. &amp;&gt; b 0 1 2 6 7 8 12 13 14</pre>	<p>If x is an atom, x{.y takes from y an interval of  x  items; beginning at the front if x is positive, ending at the tail if it is negative.</p> <p>In an <i>overtake</i> (in which the number to be taken exceeds the number of items), extra items consist of <i>fills</i>; zeros if y is numeric, a: if it is boxed, and spaces otherwise. The fill atom f is also specified by <i>fit</i>, as in { . ! . f.</p> <p>In general, if y is not an atom, x may be a list of length not more than \$\$y, and if y is an atom, it is replaced by ((#x)\$1)\$y. Element k produces (k{x){. "((\$\$y)-k) y.</p>
---	--

The following examples illustrate the use of the dyad *take*:

```
y=: i. 3 4
y;(2{.y);(5{.y);(_5{.y);(_6{. 'abcd');(2 _3{.y)
```

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

```
2 { . "1 y
0 1
4 5
8 9
6 { . 'ab'; 'cde'; 'fghi'
```

```
+-----+-----+-----+
| ab | cde | fghi | | |
+-----+-----+-----+
```

Tail

{ : \_

{ : selects the last item of its argument.	
--	--

```
l y=: a.{~ (a.i.'A') + i.4 5
ABCDE
FGHIJ
KLMNO
PQRST
```

```
f=: } : ; { :
f y
+-----+
|ABCDE|PQRST|
|FGHIJ|
|KLMNO|
+-----+
```

```
g=: } : ,.@; { :
g y
+-----+
|ABCDE|
|FGHIJ|
|KLMNO|
+-----+
|PQRST|
+-----+
```

```
h=: { . ,.@; } .
h y
+-----+
|ABCDE|
+-----+
|FGHIJ|
|KLMNO|
|PQRST|
+-----+
```

```
{ : "1 y
EJOT
```



# Map                      { ::        1        Fetch

`{ :: y` has the same boxing as `y` and its elements are the paths to each leaf (each open array).

`x { :: y` fetches a subarray of `y` according to path `x`; the selection at each level is based on `{` and, except at the last level, must result in an atom.

Map and Fetch can be modeled as follows:

```
cat  =: { @: (i.&.>) @: $
mapp =: 4 : 'if. L. y. do. (<"0 x.,&.><"0 cat y.) mapp&.> y.
      else. >x. end.'
```

```
map  =: a:&mapp

fetch=: >@({&>/)@(<"0@|.@[ , <@]) " 1 _
```

The following phrases illustrate the use of Map and Fetch:

```
] y=: 1 2 3;4 5;i.4 5
```

1	2	3	4	5	0	1	2	3	4
					5	6	7	8	9
					10	11	12	13	14
					15	16	17	18	19

```
(2;_1 _1){::y
```

The number 19

```
(_1;3 4) {::y
```

The number 19

```
{::y
```

Paths to each open array

```
{::cat L: 0 y
```

Paths to each open scalar

```
] t=: 5!:2 <'fetch'
```

An array with an interesting structure

<table border="1"> <tr><td>&gt;</td><td>@</td></tr> <tr><td colspan="2">  {   &amp;   &gt;  </td></tr> <tr><td colspan="2">       </td></tr> </table>	>	@	{   &   >				<table border="1"> <tr><td>@</td></tr> <tr><td>  .  </td></tr> <tr><td>   </td></tr> </table>	@	.		<table border="1"> <tr><td>  &lt;   @   ]  </td></tr> <tr><td colspan="2">       </td></tr> </table>	<   @   ]			" 1 _
>	@														
{   &   >															
@															
.															
<   @   ]															

Continued

Map

{ ::    1    }

Fetch

Continued

(0;2;0;0;0){:: t	Fetch the subarray corresp. to <"0 in t
(0;2;0;0;0;_1){:: t	Fetch the 0 in that
t ,&< L: 0 1 {:: t	Label each leaf with its path
< S: 0 t	The boxed leaves of t
< S: 1 {:: t	The boxed paths of t
t ,&< S: 0 1 {:: t	A 2-column table of leaves and paths
# 0: S: 0 t	The number of leaves in t

# Item Amend $m\}$ — — — Amend

<p>If <math>m</math> is numeric and <math>z=: m\} y</math>, then <math>\\$z</math> equals <math>\\$m</math>, which equals the shape of an item of <math>y</math>. The atom <math>j\{z</math> is <math>j\{(j\{m\}\}y</math>. For example:</p> <pre>y=: a.{~(a.i.'A')+i.4 5 m=: 3 1 0 2 1 y ; m ; m}y</pre> <table><tr><td>ABCDE</td><td>3</td><td>1</td><td>0</td><td>2</td><td>1</td><td>PGCNJ</td></tr><tr><td>FGHIJ</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>KLMNO</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>PQRST</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	ABCDE	3	1	0	2	1	PGCNJ	FGHIJ							KLMNO							PQRST							<p>If <math>m</math> is not a gerund, <math>x m\} y</math> is formed by replacing by <math>x</math> those parts of <math>y</math> selected by <math>m\&amp;\{</math>. Thus:</p> <pre>y; '%*(1 3;2 _1)} y</pre> <table><tr><td>ABCDE</td><td>ABCDE</td></tr><tr><td>FGHIJ</td><td>FGH%J</td></tr><tr><td>KLMNO</td><td>KLMN*</td></tr><tr><td>PQRST</td><td>PQRST</td></tr></table> <p><math>\\$x</math> must be a suffix of <math>\\$m\{y</math>, and <math>x</math> has the same effect as <math>(\\$m\{y)\\$,x</math>. Thus:</p> <pre>y; 'think' 1 2} y</pre> <table><tr><td>ABCDE</td><td>ABCDE</td></tr><tr><td>FGHIJ</td><td>think</td></tr><tr><td>KLMNO</td><td>think</td></tr><tr><td>PQRST</td><td>PQRST</td></tr></table>	ABCDE	ABCDE	FGHIJ	FGH%J	KLMNO	KLMN*	PQRST	PQRST	ABCDE	ABCDE	FGHIJ	think	KLMNO	think	PQRST	PQRST
ABCDE	3	1	0	2	1	PGCNJ																																							
FGHIJ																																													
KLMNO																																													
PQRST																																													
ABCDE	ABCDE																																												
FGHIJ	FGH%J																																												
KLMNO	KLMN*																																												
PQRST	PQRST																																												
ABCDE	ABCDE																																												
FGHIJ	think																																												
KLMNO	think																																												
PQRST	PQRST																																												

If  $m$  is a gerund, one of its elements determines the index argument to the adverb  $\}$ , and the others modify the arguments  $x$  and  $y$ :

$x (v0\`v1\`v2)\} y \quad \leftrightarrow (x v0 y) (x v1 y)\} (x v2 y)$   
 $(v0\`v1\`v2)\} y \quad \leftrightarrow (v1 y)\} (v2 y)$   
 $(\quad v1\`v2)\} y \quad \leftrightarrow (v1 y)\} (v2 y)$

For example, the following functions  $E1$ ,  $E2$ , and  $E3$  interchange two rows of a matrix, multiply a row by a constant, and add a multiple of one row to another:

```

E1=: <@] C. [ [ . E2=: f`g`[ [ . E3=: F`g`[ [
f=: {:@] * {:@] { [
F=: [: +/ (1:,{:@]) * (}:{:@] { [ ]
g=: {:@]
M=: i. 4 5
M;(M E1 1 3);(M E2 1 10);(M E3 1 3 10)

```

0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
5 6 7 8 9	15 16 17 18 19	50 60 70 80 90	155 166 177 188 199
10 11 12 13 14	10 11 12 13 14	10 11 12 13 14	10 11 12 13 14
15 16 17 18 19	5 6 7 8 9	15 16 17 18 19	15 16 17 18 19

Item Amend

u}      \_ \_ \_

Amend

u} is defined in terms of the noun case m}, the verb u applying to the argument or arguments to provide the numeric indices required by it.

For example:

x=: 100 + i. 2 4

u=: \*/@\$@] | (5: \* i. @\$@[ )

y=: i. 3 2 4

x ; y ; (x u y) ; (x u} y)

100 101 102 103	0 1 2 3	0 5 10 15	100 105 2 3
104 105 106 107	4 5 6 7	20 1 6 11	4 101 106 7
	8 9 10 11		8 9 102 107
	12 13 14 15		12 13 14 103
	16 17 18 19		16 17 18 19
	20 21 22 23		104 21 22 23

The positions selected by x u} y may be made to depend on either or both of the arguments x and y, and related adverbs can be defined for convenient use in common cases. For example:

A=: @(i. @\$@]

u=: (<0 1)&|:

x=: 'DIAG' [ y=: a. {~ (a. i. 'a') + i. 4 5

x;y;(x u A y);(x u} A} y)

DIAG	abcde	0 6 12 18	Dbcde
	fghij		fIhij
	klmno		klAno
	pqrst		pqrGt

Also see the case m} for the use of gerunds.

**Behead**

} . \_ 1 \_

**Drop**

} . drops the leading item of its argument.

$x$  } .  $y$  drops (at most)  $|x|$  items from  $y$ , dropping from the front if  $x$  is positive and from the tail if negative.

In general, if  $y$  is not an atom,  $x$  may be a list of length at most  $r = \$\$y$ , and the effect of element  $k$  is  $(k\{x\} \} . "(r-k) y$ ; if  $y$  is an atom, the result is  $(0=x)\$y$ .

```
ly=: a. {~ (a. i. 'A') + i. 4 5
```

```
ABCDE
FGHIJ
KLMNO
PQRST
```

```
f=: } . ; { .
f y
```

```
+-----+
|FGHIJ|ABCDE|
|KLMNO|
|PQRST|
+-----+
```

```
g=: } . ,.@; { .
g y
```

```
+-----+
|FGHIJ|
|KLMNO|
|PQRST|
+-----+
|ABCDE|
+-----+
```

```
(2}.y) ; (_2}.y) ; (6}.y) ; ($ 6}.y) ; (}. "1 y) ; (3}. "1 y)
```

```
+-----+-----+-----+-----+
|KLMNO|ABCDE|   |0 5|BCDE|DE|
|PQRST|FGHIJ|   |   |GHIJ|IJ|
|   |   |   |   |LMNO|NO|
|   |   |   |   |QRST|ST|
+-----+-----+-----+-----+
```

# Curtail } : \_

}:y drops the last item of y, and is equivalent to _1 }. y. Thus:	
---	--

```
l y=: a. {~ (a. i. 'A') + i. 4 5
ABCDE
FGHIJ
KLMNO
PQRST
```

```
f=: }: i {:
f y
+-----+
|ABCDE|PQRST|
|FGHIJ|
|KLMNO|
+-----+
```

```
g=: }: ,.@i {:
g y
+-----+
|ABCDE|
|FGHIJ|
|KLMNO|
+-----+
|PQRST|
+-----+
```

```
h=: { . ,.@i }.
h y
+-----+
|ABCDE|
+-----+
|FGHIJ|
|KLMNO|
|PQRST|
+-----+
```

```
}:"1 y
ABCD
FGHI
KLMN
PQRS
```

## Rank $m^n n$

The verb  $m^n n$  produces the constant result  $m$  for each cell to which it applies. The rank used is  $3 \text{ \$\&}. | . n$ . For example, if  $n=:2$ , the three ranks are  $2 \ 2 \ 2$ , and if  $n=: \ 2 \ 3$ , they are  $3 \ 2 \ 3$ . A negative rank is complementary:  $m^n (-r) y$  is equivalent to  $m^n (0>.(#\$y)-r) y$ .

Thus:

```

v=: 2 3 5 7
m=: i. 2 3
m ; (m"0 v) ; (m"1 v) ; (m"1 m)
+-----+
| 0 1 2 | 0 1 2 | 0 1 2 | 0 1 2 |
| 3 4 5 | 3 4 5 | 3 4 5 | 3 4 5 |
|       |       |       |       |
|       | 0 1 2 |       | 0 1 2 |
|       | 3 4 5 |       | 3 4 5 |
|       |       |       |       |
|       | 0 1 2 |       |       |
|       | 3 4 5 |       |       |
|       |       |       |       |
+-----+
      v m" 1 2 m
0 1 2
3 4 5

```

The verbs  $\_9:$  through  $9:$  are *constant* verbs, equivalent to  $\_9"$  through  $9"$ . For example:

```

odds=: 1: + 2: * i.
odds 5
1 3 5 7 9

```

Rank

u " n

The verb `u " n` applies `u` to each cell as specified by the rank `n`. The full form of the rank used is `3 $&. | . n .`. For example, if `n=:2`, the three ranks are `2 2 2`, and if `n=: 2 3`, they are `3 2 3`. A negative rank is complementary: `u " (-r) y` is equivalent to `u " (0>.(#$y)-r) y`.

( ] ; , ; ,"2) y=: a. {~ (a.i.'A') + i. 2 3 4

ABCD EFGH IJKL  MNOP QRST UVWX	ABCDEFGH IJKLMNOP QRSTUVWX	ABCDEFGH IJKLMNOP QRSTUVWX
--	----------------------------------	----------------------------------

( <"0 ; <"1 ; <"2 ; <"3 ,&< <"\_1) y    Boxing of ranks 0, 1, 2, 3, \_1

<table><tr><td>A</td><td>B</td><td>C</td><td>D</td></tr><tr><td>E</td><td>F</td><td>G</td><td>H</td></tr><tr><td>I</td><td>J</td><td>K</td><td>L</td></tr><tr><td>M</td><td>N</td><td>O</td><td>P</td></tr><tr><td>Q</td><td>R</td><td>S</td><td>T</td></tr><tr><td>U</td><td>V</td><td>W</td><td>X</td></tr></table>	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	<table><tr><td>ABCD</td><td>EFGH</td><td>IJKL</td></tr><tr><td>MNOP</td><td>QRST</td><td>UVWX</td></tr></table>	ABCD	EFGH	IJKL	MNOP	QRST	UVWX	<table><tr><td>ABCD</td><td>MNOP</td></tr><tr><td>EFGH</td><td>QRST</td></tr><tr><td>IJKL</td><td>UVWX</td></tr></table>	ABCD	MNOP	EFGH	QRST	IJKL	UVWX	<table><tr><td>ABCD</td><td>EFGH</td><td>IJKL</td></tr><tr><td>MNOP</td><td>QRST</td><td>UVWX</td></tr></table>	ABCD	EFGH	IJKL	MNOP	QRST	UVWX	<table><tr><td>ABCD</td><td>MNOP</td></tr><tr><td>EFGH</td><td>QRST</td></tr><tr><td>IJKL</td><td>UVWX</td></tr></table>	ABCD	MNOP	EFGH	QRST	IJKL	UVWX
A	B	C	D																																																	
E	F	G	H																																																	
I	J	K	L																																																	
M	N	O	P																																																	
Q	R	S	T																																																	
U	V	W	X																																																	
ABCD	EFGH	IJKL																																																		
MNOP	QRST	UVWX																																																		
ABCD	MNOP																																																			
EFGH	QRST																																																			
IJKL	UVWX																																																			
ABCD	EFGH	IJKL																																																		
MNOP	QRST	UVWX																																																		
ABCD	MNOP																																																			
EFGH	QRST																																																			
IJKL	UVWX																																																			

( '\*#' , "0 1 ' abcde' ) ; (+/"2 i. 2 3 4) ; (+/"\_1 i. 2 3 4)

* abcde	12 15 18 21	12 15 18 21
# abcde	48 51 54 57	48 51 54 57



## Assign Rank $m"v$ $u"v$ $mv$ $lv$ $rv$

The verbs  $m"v$  and  $u"v$  are equivalent to  $m"r$  and  $u"r$ , where  $r$  is the list of ranks of  $v$ . The results may be examined by using the *basic characteristics* adverb  $b.$  to obtain ranks.

For example:

```

      , b. 0
- - -

      %. b. 0
2 _ 2

      ravel=: , " %.

      ravel b. 0
2 _ 2

      ]y=: a. {~ (a. i. 'A') + i. 2 3 4
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

      ,y
ABCDEFGH IJKLMN OPQRST UVWX

      ravel y
ABCDEFGH IJKL
MNOPQRST UVWX

```

Do

" . 1 \_ \_

Numbers

<p>" . y executes the sentence y. If the execution results in a noun, the result of " . y is that noun; if the execution result is a verb, adverb, or conjunction, the result of " . y is an empty vector.</p>	<p>x" . y converts character array y into numbers. The shape of the result is ( { : \$y ) , n where n is the maximum number of numbers in any row. x is a scalar number used to replace illegal numbers and to pad narrow rows. In the conversion, the normal rules for numeric constants are relaxed as follows:</p> <ul style="list-style-type: none"><li>• the negative sign can be - or _</li><li>• commas within numbers are ignored</li><li>• fractions need not have a digit 0 before the decimal point</li></ul>
--	--

For example:

```
" . s=: '5 * a=: 3 + i. 6'
15 20 25 30 35 40

a
3 4 5 6 7 8

do=: " .
do t=: '3 % 5'
0.6
do |. t
1.66667
$ do ''
0

]program=: 'a=: 2^3' ,: '5*a'
a=: 2^3
5*a

do program
8 40

do 'sum=: +/'
sum 1 2 3 4
10
s ; _999" . s=: '1 2 3' , '-4 .5' ,: 'bad 3,141'
+-----+
| 1 2 3 | 1 2 3 |
| -4 .5 | _4 0.5 _999 |
| bad 3,141 | _999 3141 _999 |
+-----+
```

## Default Format      " :      \_ 1 \_      Format

<p>Default output is identical to this monadic case, providing a minimum of one space between columns. For example:</p> <pre>       ]text=: ": i. 2 5 0 1 2 3 4 5 6 7 8 9        \$ text 2 9        3 + text  domain error        3 +text        '*#' ,. text *0 1 2 3 4 #5 6 7 8 9        ": 'abcd' abcd        \$ ": '' 0 </pre>	<p><math>x":y</math> produces a literal representation of <math>y</math> in a format specified by <math>x</math>. Each element of <math>x</math> is a complex number <math>w\ j\ .\ d</math>, controlling the representation of the corresponding column of <math>y</math> as follows:</p> <p><math> w</math> specifies the width allocated; if this space is inadequate, the entire space is filled with asterisks. If <math>w</math> is zero, enough space is allocated.</p> <p><math> d</math> specifies the number of digits following the decimal point (itself included only if <math>d</math> is not zero).</p> <p>Any negative sign is placed just before the leading digit. If <math>w&gt;0</math> and <math>d&gt;0</math>, the result is right-justified in the space. Otherwise (if <math>w&lt;0</math> or <math>d&lt;0</math>), the result is put in exponential form (with one digit before the decimal point) and is left-justified except for two fixed spaces on the left (including one for a possible negative sign).</p> <p>The format <math>w+d\%10</math> (or its negation) is obsolescent; e.g. <math>\_7.2</math> instead of <math>\_7j2</math>.</p> <p>See below for boxed arguments.</p>
--	---

```

      n ; 6j2 ": n=: % i. 2 4
+-----+-----+
|      1      0.5 0.333333 |      1.00  0.50  0.33 |
| 0.25 0.2 0.166667 0.142857 | 0.25  0.20  0.17  0.14 |
+-----+-----+

      (7j2 ": -n) ; (3j2 ": n)
+-----+-----+
|      1.00  _0.50  _0.33 | _***** |
| _0.25  _0.20  _0.17  _0.14 | ***** |
+-----+-----+

```

Continued

Default Format

" :    1    Format

Continued
-----------

```
6j3 0j_6 " : 1r2 ^ 1 1000 *"1 i.5 2
1.000 9.332636e_302
0.250 8.128549e_904
0.063 7.079811e_1506
0.016 6.166381e_2108
0.004 5.370801e_2710
```

The fit conjunction and 9!:10 specify the number of digits for floating-point numbers. For example:

```
( " : ; " :!.6 ; " :!.4 ; " :!.15 ) %7
+-----+-----+-----+-----+
|0.142857|0.142857|0.1429 |0.142857142857143|
+-----+-----+-----+-----+
```

For a boxed right argument, a two-element left argument specifies position in the display, using 0, 1, and 2 for top/center/bottom, and left/center/right. See 9!:15 and 9!:16 for default displays. Moreover, a third element selects the box-drawing characters used. See 9!:6 and 9!:7.

```
x=: 2 3 $ (2 #&.> 1+i.6) $&.> 'abcdef'
( " : x) ,. (2 1 " : x)
+-----+-----+-----+-----+
|a      |bb     |ccc    ||      |bb     |ccc    |
|        |bb     |ccc    ||      |bb     |ccc    |
+-----+-----+-----+-----+
|dddd   |eeeeee |ffffff | ||      |eeeeee |ffffff |
|dddd   |eeeeee |ffffff | |dddd  |eeeeee |ffffff |
|dddd   |eeeeee |ffffff | |dddd  |eeeeee |ffffff |
|        |eeeeee |ffffff | |dddd  |eeeeee |ffffff |
+-----+-----+-----+-----+
```

# Tie            m`n      m`v      u`n      u`v

In English, a gerund is a noun that carries the force of a verb, as does the noun *cooking* in *the art of cooking*. The tie applies to two verbs to produce a gerund. Gerunds are commonly used with *insert (/)* and with *agenda (@.)*:

```

]g=: +`*
+---+
|+|*|
+---+

(g/1 2 3 4 5) ; (1+2*3+4*5)
+---+
|47|47|
+---+

```

More generally, tie produces gerunds as follows:  $u`v$  is  $au,av$ , where  $au$  and  $av$  are the (boxed noun) *atomic representations* (5! : 1) of  $u$  and  $v$ . Moreover,  $m`n$  is  $m,n$  and  $m`v$  is  $m,av$  and  $u`n$  is  $au,n$ . See Bernecky and Hui [12]. Gerunds may also be produced directly by boxing. Thus:

```

]h=: '+' ; '*'
+---+
|+|*|
+---+

h/1 2 3 4 5
47

```

The atomic representation of a noun (used so as to distinguish a noun such as '+' from the verb +) is given by the following function:

```

(ar=: [: < (,'0')"_ ; ]) '+'
+-----+
|+---+|
||0|+||
|+---+|
+-----+

*`(ar '+' )
+---+-----+
|*|+---+|
||0|+||
|+---+|
+---+-----+

```

# Evoked Gerund

$$m' : n$$

\_\_\_\_\_

This adverb is defined for three cases:

$m \leq 0$  *Append* Appends the results of the individual verbs; the ranks are the maxima over their ranks.

m ˘ : 3 *Insert* Inserts verbs between items. Equivalent to m/

m <sub>1</sub> : 6	<i>Train</i>	Result is the train of individual verbs.
--------------------	--------------	--

For example:

```

<+:`-:``%`-: 0 a=: 1 2 3 4 5
+-----+
| 2      4      6      8      10 |
| 0.5    1      1.5    2      2.5 |
| 1 0.5  0.333333  0.25  0.2 |
+-----+

(+ b.0) ; (%. b.0) ; (+`%.`:0 b.0)
+-----+
| 0 0 0 | 2 _ 2 | _ _ _ |
+-----+

(+`*`-:3 a) ; (+`*/a) ; (1+2*3+4*5)
+--+--+--+--+
| 47 | 47 | 47 |
+--+--+--+--+

(+`*`-`-: 6 a) ; ((+ * -) a)
+-----+
| _1 _4 _9 _16 _25 | _1 _4 _9 _16 _25 |
+-----+

```

# Atop $u@v$ $mv$ $lv$ $rv$

$u@v \ y \leftrightarrow u \ v \ y$ . For example, $+:@-7$ is $\_14$ (double the negation). Moreover, the monadic uses of $u@v$ and $u\&v$ are equivalent.	$x \ u@v \ y \leftrightarrow u \ x \ v \ y$ . For example, $3 \ +:@-7$ is $\_8$ (double the difference).
---	--

Because adverbs and conjunctions are (as stated more precisely in Section II E) executed before verbs, phrases involving them are commonly used in trains without parentheses. For example:

```
mean=: +/ % #
mean 1 2 3 4
2.5
```

```
f=: +:@*: +/ -:@%:      Addition table of doubled square and halved sqrt
f 1 2 3 4
2.5 2.70711 2.86603 3
8.5 8.70711 8.86603 9
18.5 18.7071 18.866 19
32.5 32.7071 32.866 33
```

Because a conjunction applies to the entity immediately to its right, expressions to the right of conjunctions commonly require parenthesization. For example:

```
g=: *:@(+/)
h=: *:@+/
g 1 2 3 4
100
h 1 2 3 4
6770404

k=: *:@+
k/ 1 2 3 4
6770404
```

Compare the behaviour of  $@$  with that of  $@:$ . They differ only in the ranks of the verbs that they produce.

## Agenda $m@.n$ $m@.v$ $mv$ $lv$ $rv$

$m@.n$  is a verb defined by the gerund  $m$  with an *agenda* specified by  $n$ ; that is, the verb represented by the train selected from  $m$  by the indices  $n$ . If  $n$  is boxed, the train is parenthesized accordingly. The case  $m@.v$  uses the result of the verb  $v$  to perform the selection.

For example:

```

dorh=: +: ` -: @. (]>9:)           Double or halve (Case statement)
dorh " 0 primes=: 2 3 5 7 11 13 17 19
4 6 10 14 5.5 6.5 8.5 9.5

_:`%:`*:@. * "0 a=: 2 1 0 _1 _2
1.41421 1 _ 1 4

g=: +`-`* [. x=: 1 2 3 [ y=: 6 5 4
(x g@.2: y)
6 10 12

(] * <:) y=: 5 4 3 2 1 0           Basis of factorial
20 12 6 2 0 0

1:`(] * <:) @. (1: < ]) "0 y       Case statement
20 12 6 2 1 1

1:`(] * $:@<:)@.(1: < ]) "0 y     Self-reference for recursion
120 24 6 2 1 1

+`-`*`% @. (1 0 3;2 0)
(- + %) (* +)

3 +`-`*`% @. (1 0 3;2 0) 4
_12.8125

```



**At**  $u@:v$  \_ \_ \_

@: is equivalent to @ except that ranks are infinite.

For example:

```
x=: 1 2 3 4
y=: 7 5 3 2
x */ @: + y
2016
```

Applies product over sums to the entire lists

```
x */ @ + y
8 7 6 6
```

Applies product over sums to each item of the list

```
+ b. 0
0 0 0
```

```
*/ @: + b. 0
_ _ _
```

```
*/ @ + b. 0
0 0 0
```

## Bond $m \& v$ $u \& n$ $\_$

$m \& v$ $y$ is defined as $m$ $v$ $y$ ; that is, the left argument $m$ is bonded with the dyad $v$ to produce a monadic function.	
--	--

For example:

```
10&^. 2 3 10 100 200
0.30103 0.477121 1 2 2.30103

base10log=: 10&^.
base10log 2 3 10 100 200
0.30103 0.477121 1 2 2.30103

sine=: 1&o.
sine o. 0 0.25 0.5 1.5 2
0 0.707107 1 _1 0
```

Similarly,  $u \& n$   $y$  is defined as  $y$   $u$   $n$ ; in other words, as the dyad  $u$  provided with the right argument  $n$  to produce a monadic function (that is, a function whose dyadic case has an empty domain). For example:

```
^&3 (1 2 3 4 5)
1 8 27 64 125

^&2 3"0 (1 2 3 4 5)
1 1
4 8
9 27
16 64
25 125
```

Use of the bond conjunction is often called *Currying* in honor of Haskell Curry.

## Compose $u \& v$ $mv$ $mv$ $mv$

$u \& v \ y \leftrightarrow u \ v \ y$ . Thus $+: \&-$ 7 is $\_14$ (double the negation). Moreover, the monads $u \& v$ and $u @ v$ are equivalent.	$x \ u \& v \ y \leftrightarrow (v \ x) \ u \ (v \ y)$ . For example, $3 \ + \&! \ 4$ is 30, the sum of factorials.
---	---

The monadic case is equivalent to the composition used in mathematics, but the dyadic case opens up other possibilities. For example:

$$3 \ + \&^{\cdot} \ 4 \\ 2.48491$$

Sum of natural logarithms

$$^{\cdot} 3 \ + \&^{\cdot} \ 4 \\ 12$$

Multiplication using natural logs

$$3 \ + \&(10 \&^{\cdot} \ ) \ 4 \\ 1.07918$$

Sum of base ten logarithms

$$10 \ ^{\cdot} 3 \ + \&(10 \&^{\cdot} \ ) \ 4 \\ 12$$

Multiplication using base ten logs

$$3 \ + \&^{\cdot} \ 4 \\ 12$$

See the related conjunction *under*  $(\&^{\cdot} \ )$ .

$$3 \ + \&^{\cdot} (10 \&^{\cdot} \ ) \ 4 \\ 12$$

Compare the behavior of  $\&$  with that of  $\&^{\cdot}$ . They differ only in the ranks of the verbs that they produce.

# Under $u \& . v$ $mv$ $mv$ $mv$

The verb  $u \& . v$  is equivalent to the composition  $u \& v$  except that the verb obverse to  $v$  is applied to the result for each cell. The obverse is normally the inverse, as discussed more fully under the power conjunction  $^:$ .

$3 + \& . ^ . 4$  Inverse of natural log is the exponential  
12

$(^ . ^: _1) (^ . 3) + (^ . 4)$   
12

$(< b), < | . b =: 1\ 2\ 3 ; 2\ 3\ 5\ 7 ; 'abcde'$

+-----+-----+-----+	+-----+-----+-----+
1 2 3   2 3 5 7   abcde	abcde   2 3 5 7   1 2 3
+-----+-----+-----+	+-----+-----+-----+

$each =: \& . >$  An adverb  
 $(< | . \& . > b), (< | . each\ b)$  Reversal under open

+-----+-----+-----+	+-----+-----+-----+
3 2 1   7 5 3 2   edcba	3 2 1   7 5 3 2   edcba
+-----+-----+-----+	+-----+-----+-----+

In mathematics, certain cases of *under* are called *dual* or, *dual with respect to*:

$f =: + . \& . - .$  Dual with respect to boolean negation  
 $f / \sim d =: 0\ 1$   
 0 0  
 0 1

$D =: \& . - .$  The adverb dual with respect to negation  
 $(+ . D / \sim d) ; (* . / \sim d) ; (= D / \sim d) ; (\sim : / \sim d)$

+-----+-----+-----+	+-----+-----+-----+
0 0   0 0   0 1   0 1	0 0   0 0   0 1   0 1
+-----+-----+-----+	+-----+-----+-----+

$DWL =: \& . ^ .$  Dual with respect to natural logarithm  
 $DAN =: \& . -$  Dual with respect to arithmetic negation  
 $(3 + DWL\ 4), (3 * 4), (3 < . DAN\ 4) , (3 > . 4)$   
 12 12 4 4

## Appose

$u \& : v$                         

$\& :$  is equivalent to  $\&$  except that the ranks of the resulting function are infinite; the relation is similar to that between  $@ :$  and  $@$ .

For example:

```
a=: 'abcd' ; 'efgh'
b=: 'ABCD' ; 'EFGH'
```

```
      a
+-----+-----+
|abcd|efgh|
+-----+-----+
```

```
      b
+-----+-----+
|ABCD|EFGH|
+-----+-----+
```

```
      a ,&:> b
abcd
efgh
ABCD
EFGH
```

```
      a ,&> b
abcdABCD
efghEFGH
```

```
      > b. 0
0 0 0
```

```
      , & > b. 0
0 0 0
```

```
      , &: > b. 0
-- -- --
```

Roll

? ? . 0 0 0

Deal

? y yields a uniform random selection from the population i . y . The random seed used begins at 7^5 (16807); it is unaffected by the use of ? . .	x ? y is a list of x items randomly chosen without repetition from i . y . The corresponding use of ? . does not affect the random seed.
--	--

Use of the fixed seed function ? . makes examples reproducible:

```
? . 6
0
```

```
? . 6 6 6 6 6 6 6 6 6
0 4 2 3 1 0 4 4
```

```
6 ? . 6
5 1 2 4 3 0
```

A random permutation

```
mean=: +/ % #
mean ? . 1000 # 6
2.497
```

```
m=: ? . 4 4 $ 9
m
1 6 4 4
1 0 6 6
8 3 4 7
0 0 4 6
```

A random matrix for experimentation

```
-/ . * m
588
```

The determinant of m

```
f=: ? .@$ % ] - 1:
<3 6 f 9
```

Random 3 by 6 table in range zero to one with resolution 9

0.125	0.75	0.5	0.5	0.125	0
0.75	0.75	1	0.375	0.5	0.875
0	0	0.5	0.75	0	0.375

The random seed (a beginning value for the pseudo-random number generator) is set by the foreign conjunction using 9!:1, and queried with 9!:0.

## Alphabet/Ace      a .      a :

a . is a list of the elements of the alphabet; it determines the *collating sequence* used in grading and sorting (/ : and \ :). The content of a . as well as the ordering of its elements may differ on different computing systems.

The number of elements in the alphabet is given by \$a ., and a 32-column display is given by an expression such as 8 32\$a . . The inclusion of certain *control* characters (such as the carriage return) and *non-printing* characters make such a display difficult to decipher, but the major alphabet is usually given by :

```
1 2 3 { 8 32 $ a .
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ .
```

The index of the carriage return is commonly 13 (as may be tested by entering 13 {a .), and the indices of the space and other characters may be determined as illustrated below:

```
a . i . 'aA +- * % '
97 65 32 43 45 42 37
```

The *ace* (a unit, from Latin *as*) is denoted by a :. It is the boxed empty list <\$0.

Certain of the characters in the ASCII alphabet are designated as *national use* characters because they are, on typewriters in various countries, displaced by certain other symbols such as the pound currency sign. These may be entered (on any keyboard) according to the following scheme:

	.	:	.	:
@	AT.	AT1.	AT2.	# NO. NO1. NO2.
\	BS.	BS1.	BS2.	] RB. RB1. RB2.
^	CA.	CA1.	CA2.	} RC. RC1. RC2.
`	GR.	GR1.	GR2.	\$ SH. SH1. SH2.
[	LB.	LB1.	LB2.	ST. ST1. ST2.
{	LC.	LC1.	LC2.	~ TI. TI1. TI2.

# Anagram Index    A.    1   0   \_    Anagram

<p>If <math>T</math> is the table of all <math>n!</math> permutations of order <math>n</math> arranged in lexical order (i.e., <math>/:T</math> is <math>i. n!</math>), then <math>k</math> is said to be the <i>anagram index</i> of the permutation <math>k\{T</math>.</p> <p><math>A.</math> applied to a cycle or direct permutation yields its anagram index: <math>A. 0\ 3\ 2\ 1</math> is 5, as are <math>A. 3\ 2\ 1</math> and <math>A.&lt;3\ 1</math> and <math>A.0;2;3\ 1</math>.</p>	<p>The expression <math>k\ A. b</math> permutes items of <math>b</math> by the permutation of order <math>\#b</math> whose anagram index is <math>k</math>.</p>
---	---

For example:

```
(A. 0 3 2 1) , (A. <3 1)
5 5

A. |. i.45
119622220865480194561963161495657715064383733759999999999

<: ! 45x
119622220865480194561963161495657715064383733759999999999

tap=: i.@! A. i.
(tap 3);(/: tap 3);({/\ tap 3);(/: {/\ tap 3)
```

Table of all permutations

0 1 2	0 1 2 3 4 5	0 1 2	0 1 5 2 4 3
0 2 1		0 2 1	
1 0 2		1 2 0	
1 2 0		2 0 1	
2 0 1		1 2 0	
2 1 0		1 0 2	

In particular,  $1\ A. b$  transposes the last two items of  $b$ , and  $_1\ A. b$  reverses the list of items, and  $3\ A. b$  and  $4\ A. b$  rotate the last three items of  $b$ . For example:

```
b=: 'ABCD'
(0 3 2 1{b);(0 3 2 1 C.b);((<3 1)C.b);(3 4 A.b)

+-----+-----+-----+-----+
| ADCB | ADCB | ADCB | ACDB |
|      |      |      | ADBC |
+-----+-----+-----+-----+

(_19 5 A. b) ; (_19 |~ ! # b)

+-----+-----+
| ADCB | 5 |
| ADCB |   |
+-----+-----+
```



# Boolean m b. \_ 0 0

If  $f$  is a dyadic boolean function and  $d = : 0\ 1$ , then  $d\ f\ / \ d$  (or  $f\ /\sim d$ ) is its complete table. For example the tables for *or*, *nor*, *and*, and *not-and* (followed by their ravels) appear as follows:

(+. / ~ ; +: / ~ ; \*: / ~ ; \*: / ~)  $d = : 0\ 1$

0	1	1	0	0	0	1	1
1	1	0	0	0	1	1	0

,&. > (+. / ~ ; +: / ~ ; \*: / ~ ; \*: / ~)  $d$

0	1	1	1	1	0	0	0	0	0	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

If ordered by their ravels, each of the sixteen possible boolean dyads can be characterized by its index  $k$ ; the phrase  $k\ b.$  produces the corresponding function. Moreover, negative indexing may be used. For example:

(7  $b.$  / ~ ; 8  $b.$  / ~ ; 1  $b.$  / ~ ; 14  $b.$  / ~ ; \_2  $b.$  / ~)  $d$

0	1	1	0	0	0	1	1	1	1
1	1	0	0	0	1	1	0	1	0

The adverb  $b.$  also applies to array arguments. For example:

(<"2) 2 0 1 | : 7 8 1 15  $b.$  / ~  $d$

0	1	1	0	0	0	1	1
1	1	0	0	0	1	1	1

The monad (as in  $m\ b.\ y$ ) is equivalent to a zero left argument (as in  $0\ m\ b.\ y$ ).

Basic Characteristics

u b. \_

u b. y gives the obverse of u if y is _1; its ranks if y is 0; and its identity function if y is 1.	
---	--

For example:

```
^ b. _1
^ .

^ b. 0
0 0 0

^ b. 1
$&1@({}.@$)

g=: +&2@(*&3@*: )
ly=: g 5
77

g ^:_1 y
5

g b. _1
%:@(%&3)@(-&2)

%:@(%&3)@(-&2) y
5

g b. 0
0 0 0
```

## Characteristic or Eigenvalues $C = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$

$C = Y$ yields the <i>characteristic, own</i> , or <i>eigen</i> values of its argument, arranged in ascending order on imaginary part within real within magnitude. An atom or list $Y$ is treated as the table $[Y]$ .	$0 = C = Y$ is a diagonal matrix with the eigenvalues $C = Y$ on the diagonal. Also, $1 = C = Y$ and $1 = Y$ are the left and right eigenvectors. If $i = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ , then $+ / -$ $i = C = Y$ is $Y$ .
---	---

Not implemented in Release 4.01.

## Cycle

$C. \quad 1 \quad 1 \quad \_$

## Permute

If  $p$  is a permutation of the atoms of  $i.n$ , then  $p$  is said to be a permutation vector of order  $n$ , and if  $n=\#b$ , then  $p\{b$  is a permutation of the items of  $b$ .

<p><math>C.p</math> yields a list of boxed lists of the atoms of <math>i.\#p</math>, called the <i>standard</i> cycle representation of the permutation <math>p</math>. Thus, if <math>p=:4 \ 5 \ 2 \ 1 \ 0 \ 3</math>, then <math>C.p</math> is <math>(,2);4 \ 0;5 \ 3 \ 1</math> because the permutation <math>p</math> moves to position 2 the item 2, to 4 the item 0, to 0 the item 4, to 5 the item 3, to 3 the item 1, and to 1 the item 5. The monad <math>C.</math> is self-inverse; applied to a standard cycle it gives the corresponding direct representation.</p> <p>A given permutation could be represented by cycles in a variety of ways; the standard form is made unique by the following restrictions: the cycles are disjoint and exhaustive (i.e., the atoms of the boxed elements together form a permutation vector); each boxed cycle is rotated to begin with its largest element; and the boxed cycles are put in ascending order on their leading elements.</p> <p><math>C.</math> is extended to non-negative non-standard cases by treating any argument <math>q</math> as a representation of a permutation of order <math>1+&gt;./i \ q</math>.</p>	<p>If <math>p</math> and <math>c</math> are standard and cycle representations of order <math>\#b</math>, then <math>p \ C. \ b</math> and <math>c \ C. \ b</math> produce the permutation of <math>b</math>. The arguments <math>p</math> and <math>c</math> can be <i>non-standard</i> in ways to be defined. In particular, negative integers down to <math>-\#b</math> may be used, and are treated as their residues modulo <math>\#b</math>.</p> <p>If <math>q</math> is not boxed, and the elements of <math>(\#b) q</math> are distinct, then <math>q \ C. \ b</math> is equivalent to <math>p \ C. \ b</math>, where <math>p</math> is the standard form of <math>q</math> that is given by <math>p=:((i.n)-.n q),n q</math>, for <math>n=: \#b</math>.</p> <p>In other words, positions occurring in <math>q</math> are moved to the tail end. If <math>q</math> is boxed, the elements of <math>(\#b) &gt;j\{q</math> must be distinct for each <math>j</math>, and the boxes are applied in succession. For example:</p> <pre>(2 1;3 0 1) C. i.5 1 2 3 0 4  (&lt;2 1) C. (&lt;3 0 1) C. i.5 1 2 3 0 4</pre>
--	---

Continued

# Cycle C. 1 1 \_ Permute

## Continued

```

q=: C. p=: 1 2 3 0 4 [ a=: 'abcde'
q ; (q C. a) ; p ; (p C. a) ; (p { a)
+-----+-----+-----+-----+
|+-----+|bcdae|1 2 3 0 4|bcdae|bcdae| | |
|3 0 1 2|4||      |      |      |      |
|+-----+|+-----+|+-----+|+-----+|
+-----+-----+-----+-----+

a ; (<0 _1) C. a
+-----+
|abcde|ebcda|
+-----+

]p=: 22 ?. 22          A random permutation of order 22
19 5 10 8 14 16 20 4 0 18 15 1 9 12 3 2 11 7 17 21 13 6

C. p          Its cycles
+-----+-----+-----+-----+
|15 2 10|16 11 1 5|21 6 20 13 12 9 18 17 7 4 14 3 8 0 19|
|+-----+|+-----+|+-----+|+-----+|
+-----+-----+-----+-----+

*./ #&> C. p          LCM of the cycle lengths
60

# ~. p&{^(i.200) i.#p
60

```

# Derivative $u \, d. \, n \, 0$

$u \, d. \, n$ is like $u \, D. \, n$ except that $u$ is treated as a rank-0 function.	
--	--

```

      *: d. 1
+:
      *: d. 2
2"0
      (1: + _3&* + *: ) d. 1
_3 2&p.
      (1: + _3&* + *: ) d. 2
2"0
      (1: + _3&* + *: ) d. 3
0"0

      ^. d. 1
%
      (^. * *: ) d. 1
(% * *: ) + ^. * +:
      (^. % *: ) d. 1
((% * *: ) - ^. * +: ) % 0 0 0 0 1&p.
      (^. @ *: ) d. 1
+: * %@*:

      *: d. _1
0 0 0 0.33333333333333331&p.
      % d. _1
^.

      *: d. 1 x=: _3 _2 _1 0 1 2 3
_6 _4 _2 0 2 4 6
      +: x
_6 _4 _2 0 2 4 6
```

## Derivative m D. n u D. n mu

$u D. n$  is the  $n$ -th derivative of  $u$ , and  $u`v D. n$  is  $u$  with an assigned  $n$ -th derivative  $v$ . For example:

```
(cube D.1;cube D.2; (cube=: ^&3"0) D.3)y=: 2 3 4
+-----+-----+-----+
|12 27 48|12 18 24|6 6 6|
+-----+-----+-----+
```

The derivative applies to constant functions, polynomials, the exponential  $e^x$ , the integral powers  $x^n$ , and those assigned by  $u`v D. n$ . It also applies to functions derived from these by addition, subtraction, multiplication, and division ( $u+v$ , etc.); by the composition  $u@v$ ; and by the inverse  $u^:_1$ . Since functions such as  $\sqrt{\phantom{x}}$  and  $-$  (negation) and  $\%:$  (square root) and  $l\&o.$  (sin) and  $6\&o.$  (cosh) may all be so derived, they are also in the domain of the derivative. Others are treated by approximation. The derivative of an arbitrary function may also be treated by a polynomial approximation, (provided by the matrix divide), or by approximations using the secant slope  $D:$ .

If the argument rank of  $u$  is  $a$  and the result rank is  $r$ , then the argument rank of  $u D.1$  is also  $a$ , but its result rank is  $r+a$ : the result of  $u D.1$  is the derivative of each atom of the result of  $u$  with respect to each element of its argument, giving what is commonly referred to as the *partial derivatives*. For example:

```
volume=: */"1 [. VOLUMES=: */\"1
(volume;volume D.1;VOLUMES;VOLUMES D.1) y
+-----+-----+-----+
|24|12 8 6|2 6 24|1 3 12|
|  |  |  |0 2 8|
|  |  |  |0 0 6|
+-----+-----+-----+

determinant=: -/ . * [. permanent=: +/ . *
( );(determinant D.1);(permanent D.1) m=: *:i.3 3
+-----+-----+-----+
| 0 1 4|_201 324 _135|2249 1476 1017|
| 9 16 25|_132 _144 36|260 144 36|
|36 49 64|_39 36 _9|89 36 9|
+-----+-----+-----+
```

The adverbs  $D=: ("0)(D.1)$  and  $VD=: ("1)(D.1)$  assign ranks to their arguments, then take the first derivative; they are convenient for use in scalar and vector calculus:

```
sin=: l&o. [. x=: 0.5p1 _0.25p1
(*/\ VD y);(sin x);(sin D x);(sin D D x)
+-----+-----+-----+
|1 3 12|1 _0.707107|0 0.707107|_1 0.707107| |
|0 2 8|  |  |  |  |
|0 0 6|  |  |  |  |
+-----+-----+-----+
```

## u D: n mu mu      Secant Slope

	<p><math>x</math> u D: 1 <math>y</math> is the <i>secant slope</i> of the function <math>u</math> through the points <math>y</math> and <math>y+x</math>. The secant slope is generalized to the case <math>x</math> u D. n <math>y</math> in the manner of the derivative D.. The argument <math>x</math> may be a list, giving several slopes.</p> <p>In the general case, each item of <math>x</math> has the shape <math>\{ . \ \\$\\$ "r \ y</math>, where <math>r</math> is the rank of <math>u</math>, therefore specifying the increment in each possible direction. An argument <math>x</math> of lower rank is extended in the usual manner. For example, <math>x=: 1e\_8</math> provides the same increment in each direction and, because of the small magnitude, yields an approximation to the derivative.</p>
--	--

```

1 log D:1 y
0.405465 0.287682 0.223144

incr=: 1 0.1 0.01 1e_8
incr log D:1/y
0.405465 0.287682 0.223144
0.487902 0.327898 0.246926
0.498754 0.332779 0.249688
0.5 0.333333 0.25

log D.1 y
0.5 0.333333 0.25
%y
0.5 0.333333 0.25

f=: +/@: *: "1 [ . g=: +/@: *: "1
(f y) ; (1 f D:1 y) ; (1 0.1 1e_8 f D:1 y)
+-----+
| 29 | 5 7 9 | 5 0.61 8e_8 |
|   |   | 50 6.1 8e_7 |
|   |   | 5e8 6.1e7 8 |
+-----+
(g y) ; (1 g D:1 y)
+-----+
| 4 13 29 | 5 0 0 |
|   | 5 7 0 |
|   | 5 7 9 |
+-----+

```





Raze In

e .

— — —

Member (In)

e.y produces a boolean result that determines for each atom of y whether its open contains an item of the raze of y.	If x has the shape of an item of y, then x e. y is 1 if x matches an item of y. In general, x e. y ↔ (#y)>y i. x.  The fit conjunction provides tolerant comparison, as in e.! .t.
--	--

For example:

```
]y=: 'abc' ; 'dc' ; 'a'
+---+---+--+
|abc|dc|a|
+---+---+--+

;y
abcdca

e. y
1 1 1 0 1 1
0 0 1 1 1 0
1 0 0 0 0 1

f=: ] e.~&>/ ;
f y
1 1 1 0 1 1
0 0 1 1 1 0
1 0 0 0 0 1

'cat' e. 'abcd'
1 1 0

]z=: 2 3$'catdog'
cat
dog

'cat' e. z
1
```

## E.      Member of Interval

	The <i>ones</i> in $x \in y$ indicate the beginning points of occurrences of the pattern $x$ in $y$ .
--	---

For example:

```

      'co' E. 'cocoa'
1 0 1 0 0

      s #~ -. '**' E. s=: 'Remove***multiple**stars.'
Remove*multiple*stars.

      ] x =: 0 1 2 ,: 2 3 4
0 1 2
2 3 4

      ] y=: 5 | i. 5 7
0 1 2 3 4 0 1
2 3 4 0 1 2 3
4 0 1 2 3 4 0
1 2 3 4 0 1 2
3 4 0 1 2 3 4

      x E. y
1 0 0 0 0 0 0
0 0 0 1 0 0 0
0 1 0 0 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0

      ($x) x&-: ;. 3 y
1 0 0 0 0 0 0
0 0 0 1 0 0 0
0 1 0 0 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0

```

## Fix m f . u f .

If  $x$  is a proverb, then  $y =: x \text{ f.}$  is equivalent to it, except that any names that occur in the definition of  $x$  are (recursively) replaced by their referents. Consequently, any subsequent change in these referents that might change the definition of  $x$  will not affect the definition of  $y$ .

If  $x$  is the name of any entity (that is, a pronoun, proverb, pro-adverb, or pro-conjunction), then ' $x$ '  $\text{f.}$  is equivalent, but with all names in its definition recursively replaced by their referents.

For example:

```
sum=: +/
mean=: sum % #
norm=: - mean
norm a=: 2 3 4 5
_1.5 _0.5 0.5 1.5
```

```
N=: norm f.
N a
_1.5 _0.5 0.5 1.5
```

```
norm
- mean
```

```
N
- (+/ % #)
```

```
sum=: -/
norm a
2.5 3.5 4.5 5.5
```

```
N a
_1.5 _0.5 0.5 1.5
```

```
adv=: norm@
*: adv
norm@*:
adv
norm@
'adv' f.
(- (-/ % #))@
'a' f.
2 3 4 5
```

## Hypergeometric m H. n 0 0 0

The conjunction `H.` applies to two numeric lists to produce a monad which is the hypergeometric function defined in Section 15 of Abramowitz and Stegun [13]; it is the limit of the dyadic case, whose left argument restricts the number of terms of the approximating series.

As discussed in Iverson [14], the conjunction is defined as follows:

```
rf=: 1 : '(,x.)"_ ^!.1/ i.@['          Rising factorial
L1=: 2 : 'x.rf %&(*) y.rf'
L2=: (i.@[ ^~ ])% (!@i.@[ )
H=: L1 (+/ . *) L2
```

For example:

```
'a b'=: 2 3 5; 6 5
a L1 b
(2 3 5"_ ^!.1/ i.@[ ) %&(*) 6 5"_ ^!.1/ i.@[
t=: 4 [ z=: 7
t a L1 b z
1 1 1.71429 4.28571

t (a H b , a H. b) z
295 295

f=: 1 H. 1
8 f i. 6
1 2.71825 7.38095 19.8464 51.8063 128.619

f i. 6
1 2.71828 7.38906 20.0855 54.5982 148.413

^ i. 6
1 2.71828 7.38906 20.0855 54.5982 148.413
```

Integers

i . 1 \_ \_

Index Of

<p>The shape of <code>i.y</code> is <code> y</code>, and its atoms are the first <code>*/ y</code> non-negative integers. A negative element in <code>y</code> causes reversal of the atoms along the corresponding axis. For example:</p> <pre>i. 5 0 1 2 3 4  i. 2 _5 4 3 2 1 0 9 8 7 6 5</pre>	<p>If <code>rix</code> is the rank of an item of <code>x</code>, then the shape of the result of <code>x i. y</code> is <code>(-rix)}.\$y</code>. Each atom of the result is either <code>#x</code> or the index of the first occurrence among the items of <code>x</code> of the corresponding <code>rix</code>-cell of <code>y</code>.</p> <p>The comparison in <code>x i. y</code> is tolerant, and <i>fit</i> can be used to specify the tolerance, as in <code>i. !. t.</code></p>
---	---

(i.4);(i.\_4);(i.2 3 4);(i.2 \_3 4);(i.'')

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

A=: 'abcdefghijklmnopqrstuvwxyz'  
(A i. 'Now');(A i. 'now');(A {~ A i. 'now')

26	14	22	13	14	22	now
----	----	----	----	----	----	-----

m=: 5 4 \$ 12{. A  
m;(m i. 'efgh');(1{m);(4{m)

abcd	1	efgh	efgh
efgh			
ijkl			
abcd			
efgh			

# i:      —      —      Index Of Last

	i: is like i. but gives the index of the <i>last</i> occurrence.
--	--

For example:

```

      1 2 3 4 1 2 3 i: 1 2 2 3 3 3 4 4 5
4 5 5 6 6 6 3 3 7

      1 2 3 4 1 2 3 i. 1 2 2 3 3 3 4 4 5
0 1 1 2 2 2 3 3 7

      (i.~ ,: i:~) 'boustrophedonic'
0  1 2 3 4 5  1 7 8 9 10  1 12 13 14
0 11 2 3 4 5 11 7 8 9 10 11 12 13 14

      (3 # i.3 4) i: (i.2 4)
2 5

      (3 # i.3 4) i. (i.2 4)
0 3

```

Imaginary

j. 0 0 0

Complex

j. y ↔ 0j1 * y	x j. y ↔ x + j. y
----------------	-------------------

For example:

j. 4  
0j4

3 j. 4  
3j4

a=: i. 3 3 a:(j. 2*a):(a j. 2*a)											
0	1	2	0	0j2	0j4	0	1j2	2j4			
3	4	5	0j6	0j8	0j10	3j6	4j8	5j10			
6	7	8	0j12	0j14	0j16	6j12	7j14	8j16			

(+ a j. 2*a):( a j. 2*a)											
0	1j_2	2j_4	0	2.23607	4.47214						
3j_6	4j_8	5j_10	6.7082	8.94427	11.1803						
6j_12	7j_14	8j_16	13.4164	15.6525	17.8885						

1 2 3 j./ 4 5 6 7  
1j4 1j5 1j6 1j7  
2j4 2j5 2j6 2j7  
3j4 3j5 3j6 3j7

j./?. 2 3 4\$1000  
131j34 755j53 458j529 532j671  
218j7 47j383 678j66 679j417  
934j686 383j588 519j930 830j846

A table of random complex numbers

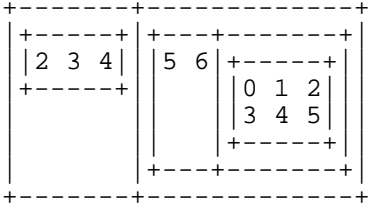


# Level L . \_

If  $y$  is open or empty,  $L.y$  is 0; if it is boxed and non-empty, the level is one plus the maximum of the levels of the opened elements.

For example:

```
]y=: (<2 3 4),<(5 6 ; <<i. 2 3)
```



```

L.y
3
L."0 y
2 3

```

# Level u L: n \_ \_ \_

The conjunction `L:` applies the verb `u` at the levels specified by `n`. The right argument `n` behaves like that of the rank conjunction in several respects:

- It may have three elements, specifying levels for the monadic, left, and right cases
- The full form of the levels used is `3$&. | .n`. For example, a 2-element list `p,q` is equivalent to `q,p,q`.
- Negative values are complementary: `u L:(-r) y ↔ u L:(0>.(L.y)-r) y`

For example:

`] y=: (<<2 3 4),<(5 6 ; <<i. 2 3)`

2 3 4	5 6	0 1 2 3 4 5
-------	-----	----------------

`+: L: 0 y`

The adverb `L:0` may be called *leaf*

4 6 8	10 12	0 2 4 6 8 10
-------	-------	-----------------

`2 # L: 0 y`

2 2 3 3 4 4	5 5 6 6	0 1 2 0 1 2 3 4 5 3 4 5
-------------	---------	----------------------------------

`2 # L: 1 y`

2 3 4 2 3 4	5 5 6 6	0 1 2 0 1 2 3 4 5 3 4 5
-------------	---------	----------------------------

## Explicit Arguments `m.` `n.`

The name `m.` denotes a *noun* left argument in an explicitly-defined adverb or conjunction, and the name `n.` denotes a *noun* right argument in an explicitly-defined conjunction. See the names `u.` `v.` `x.` `y.` and Explicit Definition (`:`).

For example:

```

    pow=: 2 : 0
      i=.0
      t=.]
      while. n.>i do.
        i=.1+i
        t=.u.@t f.
      end.
    )
    o. pow 2
o.@(o.@])
    o. pow 3
o.@(o.@(o.@]))
    o. pow 3 x=: 1
31.0063
    o.^:3 x
31.0063

    o. pow +
|value error: n.
|      n.>i

```

Uses `n.` (noun right argument)

Uses `u.` (verb left argument)

*Comment* NB.

The rest of the line following NB. is ignored.

For example:

```

text=: 'i. 3 4 NB. 3-by-4 table'
i: text
+---+---+-----+
|i.|3 4|NB. 3-by-4 table|
+---+---+-----+

". text
0 1 2 3
4 5 6 7
8 9 10 11

```

## Pi Times      0 0 0      Circle Function

$\circ.$ $y$ yields $\pi$ times $y$ . Thus $\circ. 1$ is approximately 3.14159.	The function $x\&\circ.$ is even or odd as $x$ is even or odd; $(-x)\&\circ.$ is its inverse.
---	---

$x \circ. y$	$x$	$(-x) \circ. y$	
Sqrt 1-(Sqr y)	<b>0</b>	Sqrt 1-(Sqr y)	$0\&\circ. @ (1\&\circ.) \leftrightarrow 2\&\circ.$
Sine y	<b>1</b>	Arcsine y	Radian arg/result
Cosine y	<b>2</b>	Arccos y	"
Tangent y	<b>3</b>	Arctan y	"
Sqrt (Sqr y)+1	<b>4</b>	Sqrt (Sqr y)-1	$4\&\circ. @ (5\&\circ.) \leftrightarrow 6\&\circ.$
Sinh y	<b>5</b>	Arcsinh y	Sinh is hyperbolic sine
Cosh y	<b>6</b>	Arccosh y	
Tanh y	<b>7</b>	Arctanh y	
Sqrt -(1 + Sqr y)	<b>8</b>	- Sqrt -(1 + Sqr y)	
RealPart y	<b>9</b>	y	
Magnitude y	<b>10</b>	Conjugate (+y)	
ImaginaryPart y	<b>11</b>	j. y	
AngleOf y	<b>12</b>	^j. y	

Examples:

```

rfd=: %&180 @  $\circ.$       Radians from degrees
sin=: l& $\circ.$ 
SIN=: sin@rfd
(rfd 0 90 180);(sin 0 1.5708);(SIN 0 90)
+-----+-----+
|0 1.5708 3.14159|0 1|0 1|
+-----+-----+

```

The principal domain of the inverse of a non-monotonic function is restricted. The limits on real domains of arcsine and arccos may be obtained as follows (as multiples of  $\pi$ ):

```

1p_1 * _1 _2  $\circ.$  / _1 1
_0.5 0.5
1 0

```

Roots

p. 1 1 0

Polynomial

<p>p. c ↔ (m;r) p.p.c ↔ c</p> <p>If e is a vector whose elements are all non-negative integers, then p.&lt;c,.e gives the coefficients of the equivalent polynomial: (p. &lt;c,.e)&amp;p. ↔ (&lt;c,.e)&amp;p.</p>	<p>There are three cases — coefficients; multiplier and roots; multinomial (boxed matrix of coefficients and exponents):</p> <p>c p. x ↔ +/c*x^i.#c (m;r) p. x ↔ m * */x-r (&lt;r)&amp;p. ↔ (1;r)&amp;p. (&lt;c,.e)p.&lt;y ↔ c+/ .*e*/ .(^~)y</p> <p>where m is a scalar; c and r are scalars or vectors; and e is a vector or matrix such that (\$e)-:(#c),(#y). A scalar y is extended normally.</p>
---	--

```
p. 1 0 0 1
+-----+
|1|_1 0.5j0.866025 0.5j_0.866025|
+-----+

]mr=: p. c=: 0 16 _12 2
+-----+
|2|4 2 0|
+-----+

x=: 0 1 2 3 4 5
(c p. x), ((<c,.i.4)p. x), (mr p. x),: 2*(x-4)*(x-2)*(x-0)
0 6 0 _6 0 30
0 6 0 _6 0 30
0 6 0 _6 0 30
0 6 0 _6 0 30

c=: 1 3 3 1
c p. x
1 8 27 64 125 216
(x+1)^3
1 8 27 64 125 216

bc=: !~/~i.5
bc:(bc p./ x):((i.5) ^~/ x+1)
+-----+
|1 0 0 0 0|1 1 1 1 1 1|1 1 1 1 1 1|
|1 1 0 0 0|1 2 3 4 5 6|1 2 3 4 5 6|
|1 2 1 0 0|1 4 9 16 25 36|1 4 9 16 25 36|
|1 3 3 1 0|1 8 27 64 125 216|1 8 27 64 125 216|
|1 4 6 4 1|1 16 81 256 625 1296|1 16 81 256 625 1296|
+-----+
```

Continued

# Roots $p. \ 1 \ 1 \ 0$ Polynomial

## Continued

```
c&p. d. 1 x
3 12 27 48 75 108
```

First derivative of polynomial

```
(<1 _1 ,. 5 0) p. 3
242
_1 0 0 0 0 1 p. 3
242
```

Coefficients / Exponents

```
p. <1 _1 ,. 5 0
_1 0 0 0 0 1
```

Coefficients / Exponents to Coefficients

```
c=: _1 1 2 3 [ e=: 4 2$2 1 1 1 1 2 0 2
c,.e
_1 2 1
1 1 1
2 1 2
3 0 2
```

Coefficients / Exponents

```
(<c,.e) p. <y=:2.5 _1
11.75
c +/ .* e */ .(^~) y
11.75
```

Multinomial

Note that  $(<c,.e)p.<y$  is a “proper” multinomial only if the elements of  $e$  are all non-negative integers. In general the powers are not so limited, as in the weighted sum of square root and 4-th root:

```
] t=: <2 3,.1r2 1r4
+-----+
| 2 1r2 |
| 3 1r4 |
+-----+
(t p. 16), +/ 2 3 * 16 ^ 1r2 1r4
14 14
```

The variant  $p.!s$  is a *stope* polynomial; it differs from  $p.$  in that its definition is based upon the  $stope \ ^!s$  instead of on  $^$  (power).

Primes

p: 0

The result of p: i is the ith prime. For example: p: 0 2	
--	--

p: i. 15  
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

The inverse of p: is the number of primes less than the argument, often denoted by  $\pi(n)$ :

pi=: p: ^: \_1  
pi i. 15  
0 0 0 1 2 2 3 3 4 4 4 4 5 5 6  
  
(] , pi ,: p:@pi) i.15  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
0 0 0 1 2 2 3 3 4 4 4 4 5 5 6  
2 2 2 3 5 5 7 7 11 11 11 11 13 13 17  
  
y=: (2^31)-1  
  
]a=: pi y  
105097564  
  
]b=: p: a  
2147483647  
  
b=y  
1



## Prime Factors    $q: 0 \ 0 \ 0$    Prime Exponents

$q: y$ is the list of prime factors of a positive integer argument $y$ . For example:  <pre> y=: 105600 q: y 2 2 2 2 2 2 2 3 5 5 11 */q: y 105600 \$ q: 1 0 */q: 1 1 q: b. _1 */ </pre>	<p>If <math>x</math> is positive and finite, <math>x \ q: y</math> is the list of exponents in the prime decomposition of positive integer <math>y</math>, based upon the first <math>x</math> primes; if <math>x</math> is <math>\_</math>, a sufficient number of primes is used.</p> <p>If <math>x</math> is negative and finite, <math>x \ q: y</math> is a 2-row table of the last <math> x </math> primes and exponents in the prime factorization of <math>y</math>; if <math>x</math> is <math>\_</math>, a sufficient number of primes is used. For example:</p> <pre> 2 q: 700 2 0 10 q: 700 2 0 2 1 0 0 0 0 0 0 _q: 700 2 0 2 1 </pre>
---	---

```

~.@q: 700
2 5 7
+/"1@=@q: 700
2 2 1
_q: 700
2 5 7
2 2 1

```

Distinct prime factors

Exponents in prime factorization

```

_q: !20x
2 3 5 7 11 13 17 19
18 8 4 2 1 1 1 1

```

```

y=: 100
e=: _&q:
(e ; +:&.e ; -:&.e ; %&3&.e)y
+-----+
|2 0 2|10000|10|4.64159|
+-----+

```

Completed list of exponents

Exponents, square, sqrt, cube root

Continued

**Prime Factors    q:   0   0   0   Prime Exponents**

<b>Continued</b>
------------------

```
V=: /@,:      For vectors of disparate lengths
12 (+V&.e; -V&.e; >.V&.e; <.V&.e) y      Product, quotient, LCM, GCD
+-----+-----+-----+
|1200|0.12|300|4|
+-----+-----+-----+
totient=: * -. %@ ~. &. q:      Euler's totient function

totient 700
240
+ / 1 = 700 +. i.700
240
```

**Angle** **$r \cdot 0 \ 0 \ 0$** **Polar**`r.y ↔ ^j.y``x r. y ↔ x*r. y.`

The result of `r. y` is a complex number of magnitude 1, whose real and imaginary parts are coordinates of the point on the unit circle at an angle of `y` radians. For example:

```

r. 2
_0.416147j0.909297

```

```

+. r. 2
_0.416147 0.909297

```

```

| r. 2
1

```

```

y=: 1r4 * o. i.7

```

Multiples of one-quarter  $\pi$

```

format=: 8j3&":

```

```

(format ,.y);(format +. r.y);(format +. 2 r.y)

```

0.000	1.000	0.000	2.000	0.000
0.785	0.707	0.707	1.414	1.414
1.571	0.000	1.000	0.000	2.000
2.356	−0.707	0.707	−1.414	1.414
3.142	−1.000	0.000	−2.000	0.000
3.927	−0.707	−0.707	−1.414	−1.414
4.712	0.000	−1.000	0.000	−2.000

```

3 r. _2
_1.24844j_2.72789

```

```

*. 3 r. _2
3 _2

```

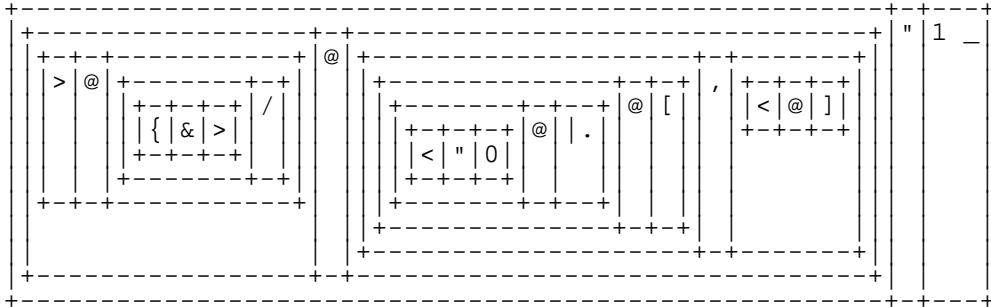
# Spread u S: n \_ \_ \_

`u S: n` produces the list resulting from applying `u` to the argument(s) at levels `n` (which has the same interpretation as the right argument of `L:`). For example, `#0:S:0 y` is the number of leaves (level 0 arrays) in `y`.

For example:

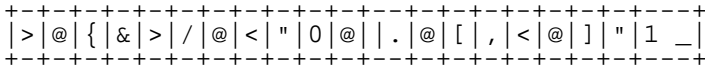
`fetch=: >@({&>/})@(<"0@|.@[ , <@]) " 1 _`  
`] t=: 5!:2 <'fetch'`

An array with an interesting structure



`< S: 0 t`

The boxed leaves of `t`



`11 { . t ( ; <@ ; ) S: 0 1 { :: t`

A 2-column table of leaves and paths

>	0	0	0
@	0	0	1
{	0	0	2
&	0	0	2
>	0	0	2
/	0	0	2
@	0	1	
<	0	2	0
"	0	2	0
0	0	2	0
@	0	2	0
.	0	2	0

-----+

## Assign Taylor m t. 0 0 0

The function `h=: u`v t.` is equivalent to `u` except that the function produced by `h t.` is `v`. For example, an explicit definition (such as the definition of `exp` below) has no associated Taylor series, but one can be assigned.

```

y=: i. 5
^y
1 2.71828 7.38906 20.0855 54.5982

^ t. y
1 1 0.5 0.1666667 0.04166667

exp=: 3 : '^ y.'
exp y
1 2.71828 7.38906 20.0855 54.5982

exp t. y
|domain error
|exp t.y

e=: exp`(%@!) t.
e y
1 2.71828 7.38906 20.0855 54.5982

e t. y
1 1 0.5 0.1666667 0.04166667

%@e t. y
1 _1 0.5 _0.1666667 0.04166667

%@^ t. y
1 _1 0.5 _0.1666667 0.04166667

```

Taylor Coefficient  $u_{t,0,0,0}$

$u_{t,y}$ is the $y$ th coefficient in the Taylor series approximation to the function $u$ . The domain of the adverb $t$ is the same as the left domain of the derivative $D$ . See the case $m_{t,}$ .	$x_{u_{t,y}}$ is the product of $(x^y)$ and $u_{t,y}$ .
--	---

For example:

```
f=: 1 2 1&p. [. g=: 1 3 3 1&p. [. x=: 10%~i=: i.8
]c=: (f*g) t. i
1 5 10 10 5 1 0 0

6.2 ":(c p. x),:(f*g) x
1.00 1.61 2.49 3.71 5.38 7.59 10.49 14.20
1.00 1.61 2.49 3.71 5.38 7.59 10.49 14.20

(c p. x)=(f*g) x
1 1 1 1 1 1 1 1

]d=: f@g t. i
4 12 21 22 15 6 1 0

(d p. x)=(f g x)
1 1 1 1 1 1 1 1

sin=: 1&o. [. cos=: 2&o.
8.4":t=: (^ t. i),(sin t. i),:(cos t. i)
1.0000 1.0000 0.5000 0.1667 0.0417 0.0083 0.0014 0.0002
0.0000 1.0000 0.0000 _0.1667 0.0000 0.0083 0.0000 _0.0002
1.0000 0.0000 _0.5000 0.0000 0.0417 0.0000 _0.0014 0.0000
```

Continued

# Taylor Coefficient     u t. 0 0 0

Continued
-----------

```

      * t
1 1  1  1 1 1  1 1
0 1  0 _1 0 1  0 _1
1 0 _1  0 1 0 _1 0

      ((sin*sin)+(cos*cos)) t. i
1 0 0 0 _2.71051e_20 0 0 0

      rf=: n%d [. n=: 0 1&p. [. d=: 1 _1 _1&p.
      ]fibonacci=: rf t. i. 20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

      2 +/\ fibonacci
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
6765

```

## Weighted Taylor $u(t) = 0 \ 0 \ 0$

The result of  $u(t) = k$  is  $(!k)*u(t) = k$ . In other words, the coefficients produced by  $t$  are the Taylor coefficients *weighted* by the factorial. As a consequence, the coefficients produced by it when applied to functions of the exponential family show simple patterns. For this reason it is sometimes called the *exponential generating function*. For example:

```

k=: i. 12
^ t: k
1 1 1 1 1 1 1 1 1 1 1 1

%t: k
1 _1 1 _1 1 _1 1 _1 1 _1 1 _1
Decaying exponential

sin=: 1&o. [. cos=: 2&o.
sinh=: 5&o. [. cosh=: 6&o.
exp=: ^ [. dec=: %t^

(exp t:,dec t:,sinh t:,cosh t:,sin t:,:cos t:) k
1 1 1 1 1 1 1 1 1 1 1 1
1 _1 1 _1 1 _1 1 _1 1 _1 1 _1
0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 _1 0 1 0 _1 0 1 0 _1
1 0 _1 0 1 0 _1 0 1 0 _1 0

```



## Taylor Approximation $u \approx T_n$

$u \approx T_n$  is the  $n$ -term Taylor approximation to the function  $u$ .

For example:

```

6.2 " : ^ T. 8 x=: 2 %~ i.8
1.00 1.65 2.72 4.48 7.38 12.13 19.85 32.23

6.2 " : ^ x
1.00 1.65 2.72 4.48 7.39 12.18 20.09 33.12

^ T. _
3 : 'g +: ^:(g ~: g@+:)^:_ ] 1 [. g=.p.&y.@((^) t.)@i.'"0
(^ = ^T._) i. 5
1 1 1 1 1

```

Compare the *conjunction*  $T.$  with the *adverb*  $t.$

## Explicit Arguments `u.` `v.`

The name `u.` denotes a *verb* left argument in an explicitly-defined adverb or conjunction, and the name `v.` denotes a *verb* right argument in an explicitly-defined conjunction. See the names `m.` `n.` `x.` `y.` and Explicit Definition (`:`).

For example:

```
pow=: 2 : 0
  i=.0
  t=.]
  while. n.>i do.
    i=.1+i
    t=.u.@t f.
  end.
)

    o. pow 2
o.@(o.@])
    o. pow 3
o.@(o.@(o.@])

    o. pow 3 x=: 1
31.0063
    o.^:3 x
31.0063

    o. pow +
|value error: n.
|    n.>i
```

Uses `n.` (noun right argument)

Uses `u.` (verb left argument)

## Explicit Arguments     $x.$     $y.$

The names  $x.$  and  $y.$  denote the left and right arguments in an explicit definition. See the names  $m.$   $n.$   $u.$   $v.$  and Explicit Definition ( $:$ ).

For example:

```
info=: 3 : 0
  (3!:0 y.);(#$y.);($y.);5!:5 <'y.'
  :
  (info x.) ,&< (info y.)
)
```

```
info i.12
+---+---+-----+
|4|1|12|i.12|
+---+---+-----+
```

```
info o.i.3 4
+---+---+-----+
|8|2|3 4|3.1415926535897931*i.3 4|
+---+---+-----+
```

```
'x' info ;:'Cogito, ergo sum.'
+-----+
|+---+---+|+---+---+-----+
|2|0||'x'| |32|1|4|<;._1 ' Cogito , ergo sum.'|
|+---+---+|+---+---+-----+
+-----+
```

Extended Precision      x :    \_    \_    \_

x: applies to real numbers and produces extended precision rational numbers. It applies to integers and produces extended integers. The implied comparison in x: is tolerant. The inverse x:^:_1 converts rationals, including extended integers, into finite precision numbers (floating point or integer).	1 x: y is the same as x: y; and 2 x: y produces the two extended integers of the numerator and denominator of the argument.
--	--

```
x: 1.2
6r5

2 x: 1.2
6 5

x: 1.2 _1.2 0 0.07
6r5 _6r5 0 7r100

x: 3j4
|domain error
| x:3j4

] pi =: o.1
3.14159

x: pi
1285290289249r409120605684

pi - 1285290289249%409120605684
1.49214e_13

x:!.0 pi
884279719003555r281474976710656
pi - 884279719003555%281474976710656
0

2 x: 1r2 3r4 5r6 _7r8
1 2
3 4
5 6
_7 8
```

See also Section II G.

## Constants

The form of a numeric constant defined and illustrated in Part I is elaborated by the use of further letters, as in  $2r3$  for two-thirds,  $2p1$  for two  $\pi$ , and  $2e3p1$  for  $2000\pi$ . The complete scheme of numeric constants obeys the following hierarchy :

.	The decimal point is obeyed first
—	The negative sign is obeyed next
e	Exponential (scientific) notation
ad ar j	Complex (magnitude and <b>angle</b> ) in <b>d</b> egrees or <b>r</b> adians ; Complex number
p x	Numbers based on <b>pi</b> ( $\circ.1$ ) and on Euler number (the <b>ex</b> ponential $^1$ )
b	Base value (using a to z for 10 to 35)

Moreover, digits with a trailing  $x$  denote an extended precision integer, and digits followed by an  $r$  followed by further digits denote a rational number. See Section II G.

For example,  $2.3$  denotes two and three-tenths and  $_{-}2.3$  denotes its negation; but  $_{-}2j3$  denotes a complex number with real part  $_{-}2$  and imaginary part  $3$ , *not* the negation of the complex number  $2j3$ . Furthermore, symbols at the same level of the hierarchy cannot be used together:  $1p2x3$  is an ill-formed number.

The following lists illustrate the main points:

```

2.3e2 2.3e_2 2j3
230 0.023 2j3

2p1 1p_1
6.28319 0.31831

1x2 2x1 1x_1
7.38906 5.43656 0.367879

2e2j_2e2 2e2j2p1 2ad45 2ar0.785398
200j_200 628.319j6.28319 1.41421j1.41421 1.41421j1.41421

16b1f 10b23 _10b23 1e2b23 2b111.111
31 23 _17 203 7.875

```

Negative integers following  $p$  and  $x$  indicate the use of reciprocals. For example,  $2p_{-}2$  is two divided by  $\pi$  squared, and  $2x_{-}2$  is two divided by the square of Euler's number.

## Constant Functions $\_9: \text{to } 9: \_ \_ \_$

The results are  $\_9$  and  $\_8$  and  $\_7$  and so on to 9.

For example:

```
x=: 1 2 3 [ y=: 4 5 6
2: y
```

2

```
x 9: y
```

9

The rank conjunction  $\_$  applies to any noun to make a constant function of specified rank. The particular constant functions illustrated above are therefore special cases of the form  $i \_$ . For example:

```
2" _ y
```

2

```
2"0 y
```

2 2 2

```
2"1 i. 2 3 4
```

2 2 2

2 2 2

```
2p1"0 y
```

6.28319 6.28319 6.28319

```
1p1 1p_1 1x1"0 y
```

3.14159 0.3183099 2.71828

3.14159 0.3183099 2.71828

3.14159 0.3183099 2.71828

A constant function with a vector result

```
0 0 0"1 y
```

0 0 0

The zero vector in a space of three dimensions

```
a=:
```

```
'abcdefghijklmnopqrstuvwxyz' "
```

```
A=:
```

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ' "
```

```
s=: ' ' "
```

```
f=: { a
```

```
f 5 8 6
```

```
fig
```

```
g=: { a,:A
```

```
g 1 5;0 8;0 6
```

```
Fig
```

## Control Structures

Control words are punctuation that determine the sequence of execution in an explicit definition. Matching control words and the enclosed sentences make up a control structure. The following control words and control structures are available:

```
break.

continue.

for.      T do. B end.
for_xyz. T do. B end.

goto_name.
label_name.

if. T do. B end.
if. T do. B else. B1 end.
if.      T do. B
elseif. T1 do. B1
elseif. T2 do. B2
end.

return.

select. T
  case. T0 do. B0
  fcase. T1 do. B1
  case. T2 do. B2
end.

try. B catch. B1 end.

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

Words beginning with **B** or **T** denote blocks, comprising zero or more simple sentences and control structures. The last sentence executed in a **T** block is tested for a non-zero value in its leading atom, and determines the block to be executed next. (An empty **T** block result or an omitted **T** block tests true.)

These control words and control structures are further detailed in the immediately following pages.

## break.

The `break.` control word is used in Explicit Definition (`:`). It may be used within a `for.`, `while.`, or `whilst.` control structure, and goes to the end of the smallest such enclosing structure. See also `continue..`

For example:

```
itn=: 3 : 0                      Inverse triangular number
  s=.0
  for_j.
    i.2e4
  do.
    if. s>:y. do. j break. end.
    s=.j+s
  end.
)

      x=: 10 100 1000 10000
      itn"0 x
5 15 46 142

      ]y=: 2&!^:_1 x
5 14.651 45.2242 141.922

      2!y
10 100 1000 10000
```



## continue.

The `continue.` control word is used in Explicit Definition (`:`). It may be used within a `for.`, `while.`, or `whilst.` control structure, and goes to the top of the smallest such enclosing structure. See also `break..`

For example:

```

sumeven=: 3 : 0
  s=.0
  for_j. i.y. do.
    if. 2|j do. continue. end.
    s=.j+s
  end.
)

    sumeven 9
20
  +/ (* -.@(2&|)) i.9
20

    sumeven 1000
249500
  +/ (* -.@(2&|)) i.1000
249500

    (sumeven = 2&!@>.&.-:) 1000
1

```

## for.

The `for.` control structure is used in Explicit Definition (`:`).

```
for.      T do. B end.
for_xyz. T do. B end.
```

The `B` block is evaluated once for each item of the array `A` that results from evaluating the `T` block. In the `for_xyz.` form, the local name `xyz` is set to the value of an item on each evaluation and `xyz_index` is set to the index of the item.

`break.` goes to the end of the `for.` control structure, and `continue.` goes to the evaluation of `B` for the next item.

For example:

```
f0=: 3 : 0
s=. 0
for. i. y. do. s=.:s end.
)

f1=: 3 : 0
s=.0
for_j. i.y. do.
  if. 2|j do. continue. end.
  s=.j+s
end.
)

comb=: 4 : 0      All size x. combinations of i.y.
z=.1 0$k=.i.#c=.1,~(y.-x.)$0
for. i.x. do. z=.:k,.&.>(-c=.:+/\c){.&.><1+z end.
)

(f0 = ])"0 ?5$100
1 1 1 1 1

(f1 = 2&!@>.&.-:)"0 ?5$100
1 1 1 1 1

3 comb 5
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

## goto\_name.

The `goto_name.` control word is used in Explicit Definition (`:`), and goes to the matching `label_name..` These control words are included to facilitate modelling of certain processes.

For example:

```
f=: 3 : 0
  if. y. do. goto_true. else. goto_false. end.
  label_true. 'true' return.
  label_false. 'false' return.
)

      fc 0
false

      fc 1
true

      fc ''
true
```

## if.

The `if.` control structure is used in Explicit Definition (`:`).

```
if. T do. B end.
if. T do. B else. B1 end.
if. T do. B elseif. T1 do. elseif. T2 do. B2 end.
```

The last sentence executed in a `T` block is tested for a non-zero value in its leading atom, determining whether the `B` block after the `do.` or the rest of the sentence is executed. An empty `T` block result or an omitted `T` block tests true.

See also the `select.` control structure.

For example:

```
plus=: 4 : 0          Addition on non-negative integers
  if. y. do. >: x. plus <: y. else. x. end.
)
```

```
  plus"0/~ i.5
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

```
sel=: 1 : 'x. # ]'
```

```
qsort=: 3 : 0
  if. 1 >: #y. do. y.
  else.
    (qsort e >sel y.),(e =sel y.),(qsort e <sel y.) [ e=.y.{~?#y.
  end.
)
```

```
  qsort 15 2 9 10 4 0 13 13 18 7
0 2 4 7 9 10 13 13 15 18
```

```
test=: 3 : 0
  if.    0<y. do. 'positive'
  elseif. 0>y. do. 'negative'
  elseif.    do. 'zero'
  end.
)
```

**Continued**

**if.**

<b>Continued</b>
------------------

```
test&.> 5 _2.71828 0
+-----+-----+-----+
|positive|negative|zero|
+-----+-----+-----+
test&.> ' ' ; 0 1 _2 ; _3 9
+-----+-----+-----+
|positive|zero|negative|
+-----+-----+-----+
```

## return.

The `return.` control word is used in Explicit Definition (`:`), and exits the verb, adverb, or conjunction.

The result of an explicit definition is the result of the last executed sentence that was not in a test block. If there is no such executed sentence, the result is `i. 0 0`.

For example:

```
f=: 3 : 0
  try.
    if. 0<:y. do. 'positive' return. else. t=.'negative' end.
  catch.
    t=.'caught'
  end.
  'it is ',t
)

f 7
positive

f _12
it is negative

f 'deipnosophist'
it is caught

(i.0 0) -: 3 : 'return.' 999
1

g=: 3 : 'if. 13=|y. do. ''triskaidekaphobia'' end.'
g 13
triskaidekaphobia

g 5
(i.0 0) -: g 5
1
```

## select.

The `select.` control structure is used in Explicit Definition (`:`).

```
select. T
  case.  T0 do. B0
  case.  T1 do. B1
  fcase. T2 do. B2
  case.  T3 do. B3
end.
```

The result `R` of `T` is compared to the elements of the result `Ri` of `Ti`, and the `Bi` block of the first `case.` or `fcase.` with a match is evaluated. Evaluation of the `select.` control structure then terminates for a `case.`, or continues with the next `B(i+1)` block for an `fcase.` (and further continues if *it* is an `fcase.`).

The comparison is `R e.&boxopen Ri` where `boxopen=:<^:(L.=0:)`, and a case with an omitted `Ti` is considered to match.

For example:

```
f0=: 3 : 0
select. y.
  case. 1;2 do. 'one two'
  case. 3   do. 'three'
  case. 4;5 do. 'four five'
  case. 6   do. 666
end.
)

f0&.> 1 2 3 4 5 6
+-----+-----+-----+-----+-----+-----+
|one two|one two|three|four five|four five|666|
+-----+-----+-----+-----+-----+-----+

(i.0 0) -: f0 7
1
```

**Continued**

# select.

Continued
-----------

```
f1=: 3 : 0
  select. y.
    case. 'a' do. i.1
    case. 'b' do. i.2
    case.      do. i.3
  end.
)

(i.1) -: f1 'a'
1

(i.1) -: f1 'a';'b'
1

(i.2) -: f1 'b'
1

(i.3) -: f1 'x'
1


f2=: 3 : 0
  t=. ''
  select. y.
    case. 1 do. t=.t,'one '
    fcase. 2 do. t=.t,'two '
    case. 3 do. t=.t,'three '
    fcase. 4 do. t=.t,'four '
  end.
)

  f2 1
one

  f2 2
two three

  f2 3
three

  f2 4
four

  '' -: f2 5
1
```



## try.

The `try. / catch.` control structure is used in Explicit Definition (`:`).

```
try. B0 catch. B1 end.
```

The `B0` block is executed and if an error occurs, the `B1` block is executed.

For example:

```
f=: 4 : 0
  try.
    try. 3+y. catch. *:x. end.
  catch.
    'x and y are both bad'
  end.
)

      13 f 7
10

      13 f 'primogeniture'
169

      'sui' f 'generis'
x and y are both bad
```

## while.

The `while.` and `whilst.` control structures are used in Explicit Definition (`:`).

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

The last sentence executed in the `T` block is tested for a non-zero value in its leading atom. If true, the `B` block is executed. The `T` block is then retested, and so on, continuing until the `T` block tests false. (An empty `T` block result or an omitted `T` block tests true.)

`whilst.` differs from `while.` only in that it skips test the first time.

`break.` goes to the end of the `while` or `whilst.` control structure and `continue.` goes to the top.

For example:

`exp =: 4 : 0`                      Exponentiation by repeated squaring

```
z=.1
a=.x.
n=.y.
while. n do.
  if. 2|n do. z=.z*a end.
  a=.*:a
  n=.<.-:n
end.
z
)
```

```
3 exp 7
2187
```

```
3 ^ 7
2187
```

```
1.1 exp 0
1
```

```
2x exp 128
340282366920938463463374607431768211456
```

## References

1. Falkoff, A.D., and K.E. Iverson, *The Design of APL*, IBM Journal of Research and Development, July, 1973.
2. Falkoff, A.D., and K.E. Iverson, *The Evolution of APL*, ACM Sigplan Notices, August 1978.
3. Iverson, K.E., *A Dictionary of APL*, ACM APL Quote-Quad, September, 1987.
4. McIntyre, D.B., *Language as an Intellectual Tool: From Hieroglyphics to APL*, IBM Systems Journal, December, 1991.
5. Iverson, K.E., *A Personal View of APL*, IBM Systems Journal, December, 1991.
6. Hui, R.K.W., *An Implementation of J*, ISI, 1992.
7. Reiter, C.A., *Fractals, Visualization, and J*, ISI, 1995.
8. Iverson, K.E., *Exploring Math*, ISI, 1996.
9. Burke, C. et al., *J Phrases*, ISI, 1998
10. McDonnell, E.E., *Complex Floor*, APL Congress 73, North-Holland/American Elsevier.
11. McDonnell, E.E., *Zero Divided by Zero*, APL76, ACM.
12. Bernecky, Robert, and R.K. W. Hui, *Gerunds and Representations*, APL91, ACM.
13. Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards Applied Mathematics Series #55, U. S. Government Printing Office, 1964.
14. Iverson, K.E., *Concrete Math Companion*, ISI, 1995.



# Acknowledgments

We are indebted to Eric Iverson for the design and implementation of locatives, control structures, and the Windows interface.



# Appendix

## Foreign Conjunction

The conjunction  $! :$  applies to integer scalar left and right arguments to produce verbs, with the exception that the case  $5! : 0$  produces an adverb. These verbs behave like any other verb: they may be assigned names, may serve as arguments to adverbs and conjunctions, and must be used with an argument even though (as in  $6! : 0 \text{ ' '}$ ) it may have no significance. Where these verbs take names as arguments, the names are boxed, as in  $4! : 55 \text{ 'a' ; 'bc'}$  to erase the names  $a$  and  $bc$ . A bracketed left argument indicates that it is optional.





## Scripts

# 0!:

[x] 0!:k y	<p>The script <i>y</i> is executed according to the digits (zero or one) in the 3-digit decimal representation of <i>k</i>, and the resulting execution log is appended to the file named <i>x</i> (if specified):</p> <table><tr><th></th><th>1st digit</th><th>2nd digit</th><th>3rd digit</th></tr><tr><td>0</td><td>From file or noun</td><td>Stop on error</td><td>Silent</td></tr><tr><td>1</td><td>From noun</td><td>Continue on error</td><td>Display</td></tr></table> <p>For example, 0!:111 <i>abc</i> executes the <i>noun abc</i>, <i>completes</i>, and <i>displays</i>.</p> <p>If <i>y</i> is 1, input is from the keyboard: (&lt;'f2')0!:1(1) files the execution log in <i>f2</i>.</p> <p>Sessions begin with (silent, stop) execution of 0!:0&lt;'profile.ijs'</p>		1st digit	2nd digit	3rd digit	0	From file or noun	Stop on error	Silent	1	From noun	Continue on error	Display
	1st digit	2nd digit	3rd digit										
0	From file or noun	Stop on error	Silent										
1	From noun	Continue on error	Display										
0!:2 y	The script <i>y</i> is expected to be a sequence of tautologies; 0!:2 <i>y</i> is like 0!:1 <i>y</i> but stops if any result is other than all 1.												
0!:3 y	Like 0!:2 <i>y</i> , but produces a 1 or 0 result according to whether the script passes (contains only tautologies) or fails.												

Files

1 ! :

Except as otherwise noted, a file may be specified by a name (such as <'sub\abc.q') or by an integer file number obtained from open (1!:21 <'sub\abc.q').

1!:0 y	<p><b>Directory.</b> y is a string of the path search expression (a boxed string is also accepted); the result is a 5-column table of the file name, modification time, size, permission, and attributes, individually boxed. For example, try 1!:0 '*.*'. The permission and attribute columns are system dependent. For example, in Windows:</p> <div>1!:0 'j.exe'</div> <div>+-----+-----+-----+-----+   j.exe   1998 2 2 14 33 46   676864   rwx   -----a   +-----+-----+-----+-----+</div> <p>Permission is a 3-letter string indicating the read, write, and execute permissions. Attributes is a 6-letter string indicating read-only, hidden, system, volume label, directory, and archive.</p>
1!:1 y	<p><b>Read.</b> y is a file name or a file number (produced by 1!:21); the result is a string of the file contents., e.g. 1!:1 &lt;'abc.q'. The following values for y are also permitted:</p> <div>1 read from the keyboard 3 (Unix only) read from standard input (stdin)</div>
x 1!:2 y	<p><b>Write.</b> x is a string of the new contents of the file; y is a file name or file number (produced by 1!:21). The following values for y are also permitted:</p> <div>2 screen output. 4 (Unix only) standard output (stdout) 5 (Unix only) standard error (stderr)</div>
x 1!:3 y	<p><b>Append.</b> Like x 1!:2 y, but appends rather than replaces</p>
1!:4 y	<p><b>Size</b></p>
1!:5 y	<p><b>Create Directory:</b> y is a (boxed) directory name</p>
[x] 1!:6 y	<p><b>Query/Set Attributes</b></p>
[x] 1!:7 y	<p><b>Query/Set Permissions</b></p>

**Continued**

## Files

# 1 ! :

### Continued

1!:11 y	<b>Indexed Read.</b> y is a list of a boxed file name (or number) and a boxed index and length. The index may be negative. If the length is elided, the read goes to the end. For example: <pre>1!:11 'abc.x';1000 20 f=: 1!:21 &lt;'abc.x' 1!:11 f,1000 20</pre>
x 1!:12 y	<b>Indexed Write.</b> Like indexed read; x specifies the string to be written
1!:20 y	<b>File Numbers and Names.</b> A 2-column table of the open file numbers and names.
1!:21 y	<b>Open.</b> Open file named y, creating it if necessary; result is a file number.
1!:22 y	<b>Close.</b> Close file named or numbered y. Any locks are released.
1!:30 y	<b>Locks.</b> A 3-column integer table of the file number, index, and length of file locks. The argument y is required but ignored.
1!:31 y	<b>Lock.</b> y is a 3-element integer vector of the file number, index, and length of the file region to be locked; the result is 1 if the request succeeded, and 0 if it did not.
1!:32 y	<b>Unlock.</b> y is a 3-element integer vector of the file number, index, and length of the file region to be unlocked.
1!:40 y	<b>Path.</b> The directory path of the J application executable file.
1!:41 y	<b>Server Path.</b> The directory path of the JDLLServer j.dll file (or '' if not JDLLServer).
1!:42 y	<b>Library Path.</b> The directory path of the directory where the J scripts are located.
1!:55 y	<b>Erase File/Directory.</b> e.g., 1!:55<'careful'

# Host

# 2! :

2!:0 <i>y</i>	<b>Host.</b> The list <i>y</i> is executed by the host system, and the result is returned. For example, 2!:0 'dir *.exe'. Not available for Windows or Macintosh.
2!:1 <i>y</i>	<b>Spawn.</b> Like 2!:0, but yields '' without waiting for the host to finish. Any output is ignored. For example, 2!:1 can be used to invoke a text-editor. Not available for Windows or Macintosh.
2!:2 <i>y</i>	<b>Host IO.</b> (Unix only.) The host command line <i>y</i> is passed to /bin/sh for processing, connecting two file numbers to the command's standard input/output. The result is a 3-element list of the process id of the task started and the file numbers associated with its standard input and output. These file numbers also appear in the result of 1!:20. In this case, instead of appearing with a name they appear with the command line, prefixed by > (standard input) or < (standard output). The files associated with the process should be closed with 1!:22 when no longer in use. See also 2!:3 for a verb to wait for processes to complete.
2!:3 <i>y</i>	<b>Wait.</b> (Unix only.) Wait for process id <i>y</i> to terminate. The result is the status code returned by the process.
2!:4 <i>y</i>	<b>Command Line.</b> When J is used as a shell, the result of 2!:4 is a boxed list representing command line. The first element is the name of the shell script itself; subsequent elements are the arguments, if any.
2!:5 <i>y</i>	<b>Getenv.</b> The value of the shell environment variable named <i>y</i> . If the named variable is undefined, the result is 0.
2!:55 <i>y</i>	<b>Terminate Session.</b> <i>y</i> is an integer return code

Conversions

3! :

3!:0 y	<b>Type.</b> The internal type of the noun y, encoded as follows:  1       boolean 2       literal 4       integer 8       floating point 16      complex 32      boxed 64      extended integer 128     rational
3!:1 y	<b>Convert to Binary Representation.</b> A string of the binary representation of a noun y. The result is machine-dependent.
3!:2 y	<b>Convert from Binary/Hex Representation.</b> Inverse of 3!:1 and of 3!:3
3!:3 y	<b>Hex Representation.</b> Like 3!:1, but the result is a literal matrix of the hexadecimal representation. For example, under Windows:  (3!:3 x); 3!:3 x,1p1 [ x=: 1 2 3 0 _1 +-----+-----+   04000000   08000000     00000000   00000000     05000000   06000000     01000000   01000000     05000000   06000000     01000000   00000000     02000000   0000f03f     03000000   00000000     00000000   00000040     ffffffff   00000000                 00000840                 00000000                 00000000                 00000000                 0000f0bf                 182d4454                 fb210940   +-----+-----+

Continued

Conversions

3 ! :

Continued	
3 ! : 4 y 3 ! : 5 y	<div><div>If ic=: 3 ! : 4 (integer conversion) and fc=: 3 ! : 5 (floating conversion), then</div><div><div>2 ic yJ integers to binary long integers</div><div>_2 ic ybinary long integers to J integers</div><div>1 ic yJ integers to binary short integers</div><div>_1 ic ybinary short integers to J integers</div><div>0 ic ybinary unsigned short integers to J integers</div><div>2 fc yJ floats to binary doubles</div><div>_2 fc ybinary doubles to J floats</div><div>1 fc yJ floats to binary short floats</div><div>_1 fc ybinary short floats to J floats</div><div>All ranks are infinite and all inverses of k&amp;ic and k&amp;fc exist.</div></div></div>

Names

4 ! :

4 ! : 0 y	<b>Name Class.</b> Class of (boxed) name: _2 invalid _1 unused 0 noun 1 adverb 2 conjunction 3 verb 6 locale
[x] 4 ! : 1 y	<b>Name List.</b> Result is a vector of boxed names belonging to the classes 0 to 3 and 6, as defined under 4 ! : 0 above. The optional left argument specifies the initial letters of names to be included.
4 ! : 3 y	<b>Scripts.</b> List of script names that have been invoked using 0 ! :
4 ! : 4 y	<b>Script Index.</b> Index (in 4 ! : 3 ' ') of the script that defined y, or _1 if y was not defined from a script.
4 ! : 5 y	<b>Names Changed.</b> 4 ! : 5 ] 0 turns off data collection; 4 ! : 5 ] 1 turns it on and produces a list of global names assigned since the last execution of 4 ! : 5.
4 ! : 5 5 y	<b>Erase.</b>
4 ! : 5 6 y	<b>Erase All.</b> Erase all names and all locales.



# Representation

# 5! :

x 5!:0	<p><b>Define.</b> 5!:0 is an adverb and provides a complete inverse of 5!:1. That is, (5!:1 &lt;'f') 5!:0 equals f for all f.</p>
5!:1 y	<p><b>Atomic.</b> The atomic representation of the entity named y and is used in gerunds. The result is a single box containing a character list of the symbol (if primitive) or a two-element boxed list of the symbol and atomic representation of the arguments (if not primitive). “Symbol-less” entities are assigned the following encodings:</p> <ul style="list-style-type: none"> <li>0 Noun</li> <li>2 Hook</li> <li>3 Fork</li> <li>4 Bonded conjunction</li> <li>5 Bident</li> <li>6 Trident</li> <li>7 Defined operator (pro-adverb or pro-conjunction)</li> </ul> <p>For example:</p> <pre> Plus=: + 5!:1 &lt;'plus' +--+  +  +--+ noun=: 3 1 4 1 5 9 5!:1 &lt;'noun' +-----+  +-----+    0 3 1 4 1 5 9    +-----+  +-----+ increment=: 1&amp;+ 5!:1 &lt;'increment' +-----+  +-----+    &amp; +-----+     +-----+ +     0 1       +-----+    +-----+   +-----+  +-----+ </pre>

Continued

Representation

5! :

Continued	
5!:2 y	<b>Boxed.</b>  nub=: (i.@# = i.~) # ] 5!:2 <'nub'  +-----+-----+-----+   +-----+-----+-----+   #   ]       +-----+-----+-----+               i.   @   #         i.   ~             +-----+-----+-----+           +-----+-----+-----+         +-----+-----+-----+ +-----+-----+-----+
5!:4 y	<b>Tree.</b> A literal matrix that represents the named entity in tree form. Thus:  5!:4 <'nub' +- i. +- @ -+- # +----+- =     +- ~ --- i. ----+- # +- ]
5!:5 y	<b>Linear.</b> The linear representation is a string which, when interpreted, produces the named object. For example:  5!:5 <'nub' (i.@# = i.~) # ]  5!:5 <'a' [ a=: o. i. 3 4 3.14159265358979324*i.3 4  lr=: 3 : '5!:5 <'y.''' lr 10000\$x' 10000\$x'
5!:6 y	<b>Paren.</b> Like the linear representation, but is fully parenthesized.  5!:6 <'nub' ((i.@#) = (i.~)) # ]

For all but the noun case, the default displays established by 9!:3 provide convenient experimentation with all representations. For example, 9!:3 (6 4 2) specifies that the paren, tree, and boxed representations are all to be displayed.

# Time

# 6!:

6!:0 y	<b>Current.</b> The current time in order YMDHMS (with fractional seconds):  <pre>6!:0 '' 1998 3 10 23 21 14.468</pre>
6!:1 y	<b>Session.</b> Seconds since start of session
[x] 6!:2 y	<b>Execute.</b> Seconds to execute sentence y (mean of x times with default once). For example:  <pre>a=:?50 50\$100 6!:2 '%.a' 0.091 10 (6!:2) '%.a'      Mean time of 10 executions 0.0771 ts=: 6!:2 , 7!:2@]    Time and space ts '%.a' 0.08 369920</pre>
6!:3 y	<b>Delay.</b> Delay execution for y seconds. For example, 6!:3 (2.5)

Space

7! :

7!:0 y	<b>Current.</b> Space currently in use.
7!:1 y	<b>Session.</b> Total space used since start of session.
7!:2 y	<b>Execute.</b> Space required to execute sentence y. For example: <div>7!:2 ' ,/x' [ x=: 3000 2 5\$'kerygmatic' 33888</div>
7!:3 y	<b>Free Space.</b> Information on the state of the J memory manager, currently a 2-column table of the block sizes and number of free blocks for each size. The definition of the result of 7!:3, as well as the availability of 7!:3 itself, are subject to change. For example: <div>7!:3 '' 32 950 64 1135 128 554 256 157 512 74 1024 40 2048 7 4096 14 8192 1 16384 1 32768 1</div>
7!:4 y	<b>Release Space.</b> Release unused memory from the J memory manager. Use of 7!:4 can have detrimental performance effects and such use is expected to be required only rarely. The definition of the effects of 7!:4, as well as the availability of 7!:4 itself, are subject to change.

See also 9!:20 and 9!:21.

## Global Parameters

# 9 ! :

9 ! : ( 2 \* n ) queries a parameter and (if available) 9 ! : ( 1 + 2 \* n ) sets it.

9 ! : 0 y 9 ! : 1 y	<b>Random Seed.</b> Queries and sets the random seed used in pseudo-random number generation in the verb ? (Roll • Deal). The initial value is 7^5.
9 ! : 2 y 9 ! : 3 y	<b>Default Displays.</b> The representation(s) to used for default displays of non-nouns. The representations are as defined in 5 ! : n: 1 atomic, 2 boxed, 4 tree, 5 linear, 6 paren.
9 ! : 4 y 9 ! : 5 y	<b>Input Prompt.</b> The initial value is ' ' (3 spaces).
9 ! : 6 y 9 ! : 7 y	<p><b>Box-Drawing Characters.</b> An eleven-column table whose leading row is used in default display (and in the monad format " : ). Any row may be invoked by the index j in the dyad ( h , v , j ) " : x, where h and v control the horizontal and vertical positions within the boxes. The initial settings of six rows are:</p> <pre>           9 ! : 6 ' ' +++++++   -      Normal +++++++   -      7-bit ASCII +++++++   -      Spaces +++++++      Normal less   -                 -      Only   - .....      Dots </pre>
9 ! : 8 y 9 ! : 9 y	<b>Error Messages.</b> e.g. replace English messages (default) by French.
9 ! : 10 y 9 ! : 11 y	<b>Print Precision.</b> The print precision in default output (initially 6). The precision in particular cases can be set using fit, thus: " : ! . p

Continued

Global Parameters

9 ! :

Continued	
9 ! :12 y	<b>System Type.</b> 0 PC; 1 PC386; 2 Windows; 3 Macintosh; 4 Unix; 5 OS2; 6 Windows32 (Windows95 or Windows NT); _1 Other
9 ! :14 y	<b>J Version.</b> For example: 9 ! :14 ' ' ' ' ' ' 4.01/1998-03-15/10:24
9 ! :16 y 9 ! :17 y	<b>Boxed Display Positioning.</b> y is r , c specifying row and column positioning: 0 (top, left); 1 (centre); 2 (bottom, right)
9 ! :18 y 9 ! :19 y	<b>Comparison Tolerance.</b> Queries and sets the comparison tolerance. See Equal (=). The tolerance in particular cases can be set using fit, thus: = ! . t.
9 ! :20 y 9 ! :21 y	<b>Memory Limit.</b> An upper bound on the size of any one memory allocation. The memory limit is initially infinity.
9 ! :24 y 9 ! :25 y	<b>Security Level.</b> The security level is either 0 or 1. It is initially 0, and may be set to 1 (and can not be reset to 0). When the security level is 1, executing Window driver commands and certain foreigners ( ! :) that can alter the external state cause a “security violation” error to be signalled. The following foreigners are prohibited: dyads 0 ! :n, 1 ! :n except 1 ! :40, 1 ! :41, and 1 ! :42, 2 ! :n, 8 ! :n, 14 ! :n, 15 ! :n, and 16 ! :n.

## Window Driver

**11! :**

11!:0 y

**Window Driver.** See help files.

Debug

13! :

See Section II.J and the script `system\main\debug.ijs`.

13!:0 y	<b>Reset.</b> Reset stack and disable (0) or enable (1) suspension. Nearly all the facilities in the 13!: family require that suspension enabled; all the examples below assume that suspension is enabled: 13!:0 ]1
13!:1 y	<b>Display Stack</b> See also 13!:18.
13!:2 y	<b>Query Stops.</b>
13!:3 y	<b>Set Stops.</b> Explicit stops are requested by name and line number in the argument y, which contains zero or more stop specifications separated by semicolons. Each stop specification indicates a name, line numbers (if any) for the monadic case, a colon, and line numbers (if any) for the dyadic case. An asterisk indicates “all”, and a tilde indicates “except for”. For example:  13!:3 'f 0' f monadic line 0 13!:3 'f :2' f dyadic line 2 13!:3 'f 0 2:1' f monadic 0 2, dyadic 1 13!:3 'f 0; g :*' f monadic 0 and g all dyadic 13!:3 '* 0:0' monadic 0 and dyadic 0 13!:3 'a* **;* ~ab* **;' All monadic and dyadic whose names begin with a, except for any beginning with ab  f=: 3 : 0 10 11 : 20 ) 13!:3 'f 1:0' Stop at f monad line 1, dyad line 0 f ''  stop: f   11  f[1] 13!:0 ]1 Clear stack and enable suspension 3 f 4  stop: f   20  f[:0]

Continued



# Debug

# 13! :

Continued

13!:4 y	<p><b>Run Again.</b> Resume execution at the current line. For example:</p> <pre> g=: 3 : ('t=. 2*y.'; '1+t') 3 4,g 'abc'  domain error: g  t=.2      *y.  g[0]        y. abc       y.=. 25       13!:4 '' 3 4 51 </pre> <p>six-space indent indicates suspension Local value of y. Redefine local value of y. Run again</p>
13!:5 y	<p><b>Run Next.</b> Resume execution at the next line. For example:</p> <pre> h=: 3 : ('t=. 2 3*y.'; '1+t') 3 4,h 5 6 7  length error: h  t=.2 3      *y.  h[0]        t=. 99        13!:5 '' 3 4 100 </pre> <p>six-space indent indicates suspension Run next</p>
13!:6 y	<p><b>Exit and Return.</b> Exit the verb/adverb/conjunction at the top of the stack, returning result y. For example:</p> <pre> g=: 3 : ('t=. 2*y.'; '1+t') 3 4,g 'abc'  domain error: g  t=.2      *y.  g[0]        13!:6 [9 3 4 9       h=: 2&amp;*       3 4,h 'abc'  domain error: h  h[0]        13!:6 [97 3 4 97 </pre> <p>Exit g with result 9 Exit h with result 97</p>

Continued

Debug

13! :

Continued	
13!:7 y	<b>Continue.</b> Resume execution at line number y
[x] 13!:8 y	<b>Signal.</b> Signal error number {.,y (see 9!:8) with optional text x
[x] 13!:9 y	<b>Rerun.</b> Resume execution by rerunning the verb/adverb/conjunction on the top of the stack with the specified arguments. Thus:  plus=: + plus/'abc'  domain error: plus  plus[:0] 13!:13 '' See below re interpretation of stack +-----+  plus 3 0 3 + +---+             b c              +---+   +-----+ 2 (13!:9) 3 Rerun, getting another error  domain error: plus  plus[:0] 13!:13 '' Note left and right args ('a' and 5) +-----+  plus 3 0 3 + +---+             a 5              +---+   +-----+ 1 (13!:9) 5 Rerun 6
[x]13!:10 y	<b>Rerun Execute.</b> Rerun with specified executed arguments
13!:11 y	<b>Error Number.</b> Last error number
13!:12 y	<b>Error Message.</b> Last error message

Continued

# Debug

# 13! :

## Continued

13!:13 y

**Stack.** If suspension is enabled, 13!:13 '' produces an 8-column matrix of information on the verbs/adverbs/conjunctions on the stack:

- 0 Name
- 1 Error number or 0 if not in error
- 2 Line number
- 3 Name class: 3, 1, or 2, denoting verb, adverb, or conjunction
- 4 Linear representation of the entity
- 5 The name of the defining script
- 6 Argument(s) individually boxed
- 7 Locals as a 2-column matrix of name and value

In the last two columns, nouns are included *per se*, and verb, adverb, and conjunction are represented by linear forms. For example:

```
mean=: sum % #
sum=: plus/
plus=: 4 : 'x.+y.'
mean 'abcd'
|domain error: plus
|  x.      +y.
|plus[:0]
```

13!:13 '' Note tacit definitions have no locals

plus	3	0	3	4 : 'x.+y.'	c d	x. c	
					y. d		
sum	0	0	3	plus/	abcd		
mean	0	0	3	sum % #	abcd		

## Continued

Debug

13! :

Continued

13!:14	Query Latent Expression												
13!:15	Set Latent Expression. The latent expression is executed when execution is about to be suspended; error messages are suppressed; any continuation must be programmed in the latent expression.												
13!:16 y	Trace. See below.												
13!:17 y	Query . Is suspension enabled? (Set by 13!:0)												
13!:18	Stack Text. Like 13!:1, but gives the stack as a literal matrix.												
<p>The conjunction 13!:16 controls tracing.</p> <pre>u 13!:16 n (r,c) 13!:16 n</pre> <p>The right argument <i>n</i> specifies the maximum level of function call to be traced: 0 means no trace; 1 means immediate execution only; _ means trace everything; etc. The left argument can be a verb to be used for displaying arrays in the trace (and is not itself traced during tracing). It may also be integers <i>r</i>, <i>c</i>, whence the system default display is used, clipped to <i>r</i> rows and <i>c</i> columns. (Two numbers suffice as clipping parameters because the output of an <i>n</i>-dimensional array is 2-dimensional on the screen.) Finally, it may be the empty vector, whence the current trace level and display controls are shown. The result is <i>i</i>.0 0. For example:</p> <table><tr><td>trace=: 13!:16</td><td></td></tr><tr><td>lr   =: 3 : '5!:5&lt;'y.'</td><td>Linear display of an array</td></tr><tr><td>_ _ trace _</td><td>Trace everything; display everything</td></tr><tr><td>9 trace 1</td><td>Trace immediate execution only; display maximum of 9 rows</td></tr><tr><td>": trace n</td><td>Same as _ _ trace n</td></tr><tr><td>lr trace n</td><td>Linear display of trace output</td></tr></table>		trace=: 13!:16		lr   =: 3 : '5!:5<'y.'	Linear display of an array	_ _ trace _	Trace everything; display everything	9 trace 1	Trace immediate execution only; display maximum of 9 rows	": trace n	Same as _ _ trace n	lr trace n	Linear display of trace output
trace=: 13!:16													
lr   =: 3 : '5!:5<'y.'	Linear display of an array												
_ _ trace _	Trace everything; display everything												
9 trace 1	Trace immediate execution only; display maximum of 9 rows												
": trace n	Same as _ _ trace n												
lr trace n	Linear display of trace output												

Continued

# Debug

# 13! :

## Continued

Tracing provides information on results *within* a line; the action labels, “0 monad”, “1 monad”, “9 paren”, etc., are from the parse table in Section II.E, and reflect the activities of the interpreter with high fidelity.

```

_ _ (13!:16) _
i.4
----- 0 monad -----
i.
4
0 1 2 3
=====
0 1 2 3
(i.2 4) +/ .* *: 5 * 10 20 30 40
----- 2 dyad -----
5
*
10 20 30 40
50 100 150 200
----- 0 monad -----
i.
2 4
0 1 2 3
4 5 6 7
----- 9 paren -----
0 1 2 3
4 5 6 7
----- 3 adverb -----
+
/
+/
----- 4 conj -----
+/
.
*
+/ .*

```

Trace everything; display everything

action  
verb  
argument  
result  
end of parse  
result of immediate exec.

left argument  
verb  
right argument  
result

verb  
argument  
result

## Continued

Debug

13! :

Continued

----- 1 monad -----	Words to the left of *: are parsed
*:	first because an operator may “grab”
50 100 150 200	a verb.
2500 10000 22500 40000	
----- 2 dyad -----	
0 1 2 3	left argument
4 5 6 7	
+/.*	verb
2500 10000 22500 40000	right argument
175000 475000	result
=====	end of parse
175000 475000	result of immediate execution

## Data Driver

# 14! :

See help files and the script `system\main\dd.ijs`.

14!:0 y	<b>Connect</b>
14!:1 y	<b>Disconnect</b>
x 14!:2 y	<b>SQL</b>
14!:3 y	<b>Fetch</b>
x 14!:4 y	<b>Columns</b>
14!:5 y	<b>Column Names</b>
14!:6 y	<b>Source Names</b>
x 14!:7 y	<b>Select</b>
14!:8 y	<b>End</b>
14!:9 y	<b>Error</b>
14!:10 y	<b>Begin Transaction</b>
14!:11 y	<b>Commit Transaction</b>
14!:12 y	<b>Rollback Transaction</b>
14!:13 y	<b>Table</b>
14!:14 y	<b>Fetch Columns</b>
14!:15 y	<b>Row Count</b>

## Dynamic Link Library

# 15! :

See help files and the script `system\main\dll.ijc`.

x 15!:0 y	<b>Call DLL Function</b>
15!:1 y	<b>Memory Read</b>
x 15!:2 y	<b>Memory Write</b>
15!:3 y	<b>Allocate Memory</b>
15!:4 y	<b>Release Memory</b>



## Sockets

## 16! :

See help files and the script `system\main\socket.ijs`.

16!:0 y	<b>Socket</b>
16!:1 y	<b>Receive</b>
x 16!:2 y	<b>Send</b>
16!:3 y	<b>Receive from</b>
x 16!:4 y	<b>Send to</b>
16!:5 y	<b>Close</b>
16!:6 y	<b>Connect</b>
16!:7 y	<b>Bind</b>
16!:8 y	<b>Listen</b>
16!:9 y	<b>Accept</b>
16!:10 y	<b>Select</b>
16!:11 y	<b>Get Options</b>
16!:12 y	<b>Set Options</b>
16!:13 y	<b>IO Control</b>
16!:14 y	<b>Get Host Name</b>
16!:15 y	<b>Get Peer Name</b>
16!:16 y	<b>Get Socket Name</b>
16!:17 y	<b>Get Host by Name</b>
16!:18 y	<b>Get Host by Address</b>
16!:19 y	<b>Get Sockets</b>
16!:20 y	<b>Async</b>
16!:21 y	<b>Clean up</b>

Regular Expression

17! :

See help files and the script `system\main\regex.ijs`.

x 17!:0 y	<b>Match.</b> Find first match
x 17!:1 y	<b>Matches.</b> Find all matches
17!:2 y	<b>Compile.</b> Compile pattern, returning handle
17!:3 y	<b>Free.</b> Release compiled pattern
17!:4 y	<b>Handles.</b> All pattern handles
17!:5 y	<b>Info.</b> Number of sub-expressions and pattern
17!:6 y	<b>Error.</b> Error text

## Locales

## 18! :

See also Section II.I and the “Locales” lab under menu item Studio|Labs... |Locales.

18!:0 y	<p><b>Name Class.</b> Give the name class of the locale named y, with 0 for named, 1 for numbered, _1 for non-existent, and _2 for illegal name. Thus:</p> <pre>18!:0 ;:'base j z 45bad asdf 0' 0 0 0 _2 _1 1</pre>
[x] 18!:1 y	<p><b>Name List.</b> Give the names of named (0) or numbered (1) locales. The optional left argument specifies the initial letters of names. Thus:</p> <pre>18!:1 [0          All named locales +-----+-----+  base j jcfg jnewuser newuser z  +-----+-----+  asdf_bb_=: 'sesquipedalian'  'jb' 18!:1 [0      All named locales beginning in j or b +-----+-----+  base bb j jcfg jnewuser  +-----+-----+  18!:3 ''          Create a numbered locale +-+  0  +-+  18!:1 i.2          All named and numbered locales +-----+-----+  0 base j jcfg jnewuser newuser z  +-----+-----+</pre>

Continued

Locales

18! :

Continued	
[x] 18!:2 y	<p><b>Path.</b> The monad gives the (search) path for locale y; the dyad sets the path for locale y to x. The path of a locale is initially , &lt; , ' z ' , except that locale z has an empty path initially. If a name sought in locale f is not found in f, then it is sought in the locales in the path of f (but not searching <i>their</i> paths). For example:</p> <pre>(;:'a cd b') 18!:2 &lt;'f' 18!:2 &lt;'f' ++-----+  a cd b  ++-----+ The path of locale f is set to a, cd, and b.</pre>
18!:3 y	<p><b>Create.</b> Create a previously-unused numbered locale and return its name.</p> <pre>18!:3 ''          Create a numbered locale ++  0  ++ 18!:3 ''1         Create another one ++  1  ++ 18!:1 [1          Names of numbered locales ++-----+  0 1  ++-----+</pre>

Continued

# Locales

# 18! :

## Continued

18!:4 y	<b>Switch Current.</b> Switch the current locale to y. Initially the current locale is base.
18!:5 y	<b>Current.</b> The name of the current locale. For example: <pre> 18!:5 '' +-----+  base  +-----+ </pre>
18!:55 y	<b>Erase.</b> Erase locale y (once it finishes execution). A numbered locale, once erased, may not be reused; a named locale may be reused at will.

Numerical Functions

128! :

128!:0 y	<p><b>QR.</b> Produces the QR decomposition of a complex matrix <code>y</code> (in the domain of matrix inverse <code>%.</code>), an Hermitian matrix and a square upper triangular matrix, individually boxed.</p> <p><code>x=: +/ . *</code> Matrix product <code>A=: j./?. 2 7 4\$10</code> A random complex matrix <code>\$A</code> <code>7 4</code> <code>'Q R'=: 128!:0 A</code> <code>\$Q</code> <code>7 4</code> <code>\$R</code> <code>4 4</code> <code>&gt;./ ,(=i.4) - (+ :Q) x Q</code> Q is Hermitian <code>6.33846e_16</code> <code>0~:R</code> R is upper triangular <code>1 1 1 1</code> <code>0 1 1 1</code> <code>0 0 1 1</code> <code>0 0 0 1</code> <code>A -: Q x R</code> <code>1</code></p>
128!:1 y	<p><b>R Inv.</b> Invert square uppertriangular matrix.</p>

# Index

## —A—

Abramowitz, 261  
**Ace**, 210  
actuarial work, 91  
Addition table, 32  
*addition tables*, 67  
Adverb, 94  
adverbs, 1  
Adverbs, 67  
ADVERBS, 98  
**Adverse**, 159  
agenda, 142  
*agenda*, 200  
**Agenda**, 203  
AGENDA, 45  
*agreement*, 127  
**Agreement**, 98  
alphabet, 93, 177  
**Alphabet**, 210  
ALPHABET, 66  
AMBIVALENCE, 4  
*ambivalent functions*, 1  
**Amend**, 190, 191  
**Anagram**, 211  
**Anagram Index**, 211  
*and*, 98, 212  
**And**, 128  
**Angle**, 128, 237  
anonymously, 47  
**Antibase**, 170  
**Antibase Two**, 170  
APL, 93  
**Append**, 160  
*Append*, 201  
appending zeros, 141  
**Appose**, 208  
approximate, 86  
approximation, 88, 134  
arccos, 231  
arccosh, 231  
*arcs*, 83  
arcsine, 231

arcsinh, 231  
arctan, 231  
arctanh, 231  
area, 81  
argument, 13, 94  
argument rank, 218  
Arithmetic mean, 11  
array arguments, 212  
arrays, 1, 95  
ascending order, 177, 215  
**Assign Rank**, 196  
assignment, 1  
asterisks, 198  
**At**, 204  
*atom*, 95, 224  
*atomic encoding*, 114  
*atomic representation*, 52, 200, 274  
**Atop**, 202  
average, 11  
axis, 150, 162  
*axis*, 95

## —B—

**B** block, 249  
backward difference, 178  
barcharts, 71  
**Base Two**, 168, 169  
*base-x logarithm*, 137  
*basic characteristics*, 196  
basic characteristics, 58  
**Basic Characteristics**, 213  
**Behead**, 192  
Bernecky, 200, 261  
*bidents*, 102  
binary representation, 170  
**Binary Representation**, 271  
binomial coefficients, 8, 32, 48, 171, 232  
*block*, 155  
**Bond**, 205  
BOND CONJUNCTION, 15  
Boole, 2, 3  
boolean, 83, 125, 131, 146

**Boolean**, 212  
 boolean dyads, 212  
 BORDERING A TABLE, 68  
**Box**, 114  
 box-drawing characters, 199, 278  
 boxed, 13, 95, 163, 203  
 Boxed binomials, 48  
 boxed cycle, 215  
 boxed display positioning, 199, 279  
 boxed elements, 52, 215  
 boxed representation, 153, 275  
 boxed words, 166  
 boxifopen, 163  
 boxing, 200  
 boxopen, 257  
 break ., 155, 250, 252, 260  
 Burke, 261  
 by, 67

## —C—

Cap, 12, 101, 183  
 capped fork, 101  
 Cartesian product, 59, 184  
*case statement*, 45, 257  
 Case statement, 203  
*case .*, 257  
**Catalogue**, 184, 185  
*catch .*, 155, 259  
 catenate, 162  
 Ceiling, 119  
 cells, 42, 95  
 Centre on mean, 11  
**Characteristic**, 214  
**Circle**, 231  
*class*, 100  
 class of a name, 173  
 CLASSIFICATION, 71, 74  
*closed*, 86  
*closure*, 83, 84  
*coefficients*, 86, 90  
 Coefficients from roots, 88  
 collating sequence, 177, 180, 210  
 collinear, 134  
 colon, 94  
**Combinations**, 171, 252  
**Comment**, 230  
 common rank, 160

common shape, 118  
 commutative, 8  
*commute*, 144  
 COMPARATIVES, 99  
 comparison, 224  
 complementary, 228  
 complementary rank, 43, 195  
*complete*, 71  
 Complete classification table, 72  
 complex, 86, 115, 124, 167, 198  
**Complex**, 226  
 complex number, 127, 136, 237  
 complex roots, 90  
 complex sentence, 65  
**Compose**, 206  
 composition, 86, 98, 206, 207  
 COMPOSITIONS, 76, 77  
 computer programmers, 40  
 conjugate, 66  
**Conjugate**, 124  
 Conjunction, 67, 94  
 conjunctions, 1, 76  
 CONJUNCTIONS, 98  
*connection matrix*, 83  
*constant function*, 123, 142  
**Constant Functions**, 248  
 constant result, 194  
*constant verbs*, 194  
**Constants**, 247  
*continue .*, 155, 251, 252, 260  
 continued fractions, 79  
**Control Structures**, 154, 249  
*control words*, 154, 249  
**Conversions**, 271  
 convert arguments, 104  
 convolution, 176  
**Coordinate Geometry**, 26  
 copula, 93, 142  
 Copula, 67, 94  
**Copulas**, 113  
**Copy**, 167  
 correlation, 176  
 cos, 152  
 cosh, 152, 231  
 cosine, 231  
*crosses*, 144  
*cube*, 136  
 Curry, 205



**Curtail**, 193  
**Customize**, 172  
**Cut**, 164, 165  
 Cut conjunction, 80  
 cycle, 211  
**Cycle**, 215, 216  
*cycle function*, 52  
*cycle representation*, 52  
 cyclic, 141  
 cyclical extension, 164, 174, 175, 178, 179

## —D—

Data Driver, 288  
 De Morgan, 2  
**Deal**, 209  
 Debug, 281  
 decimal point, 198  
 decrement, 142  
**Decrement**, 117  
 default displays, 278  
**Default Format**, 198, 199  
 Default output, 198  
 define, 274  
 DEFINED ADVERBS, 31  
*definition*, 13  
 DEFINITIONS, 111  
 degenerate, 134  
 Degenerate, 134  
 degenerate cases, 98  
 derivative, 86, 89, 90, 137  
**Derivative**, 217, 218, 219  
*derived*, 94  
 descending order, 180  
 determinant, 81  
 Determinant, 26, 151  
**Dex**, 182  
 diagonal, 150  
 diagonals, 176  
 dictionaries, 19  
 difference calculus, 91  
 digits, 93  
 direct permutation, 211  
 DIRECTED GRAPHS, 83  
*disjoint*, 71, 215  
 DISJOINT CLASSIFICATION, 72  
 Displacement, 26  
 displacements, 81

displays, 13  
 DISTANCE, 85  
 distinct items, 145  
**Divided by**, 133  
 Divisibility, 70  
 Divisibility table, 26  
 DLL, 289  
*do*, 27, 33  
**Do**, 197  
*do .*, 155, 252, 254, 257, 260  
**Dot Product**, 151  
**Double**, 126  
**Drop**, 192  
 dual, 115, 119, 125, 126, 129, 146  
 Dual, 207  
 dyad, 205  
**Dyad**, 97  
*dyadic*, 97  
 dyadic-only, 153  
 Dynamic Link Library, 289

## —E—

**e**, 93, 136  
 each, 57  
 ebar, 221  
**Eigenvalues**, 214  
 eigenvectors, 214  
 elements, 132  
*else .*, 155, 254  
*elseif .*, 254  
 empty, 97  
 empty list, 141  
 empty **T** block result, 249, 254, 260  
*end .*, 155, 252, 254, 257, 259, 260  
 English phrases, 33  
 epsilon, 220  
 equal elements, 177, 180  
 equality, 172  
 erase, 273  
 erase all, 273  
 error, 159, 259  
 error messages, 278  
 Errors, 94  
**ERRORS**, 109  
 Euler number, 247  
*Euler's number*, 136  
**Even**, 152

*even* function, 38  
 evoke, 113  
**Evoke**, 143  
**Evoke Gerund**, 201  
*exclusive-or*, 146  
*execute*, 27, 33, 197  
 execution, 267  
 EXECUTION, 99  
 EXERCISES, 93  
 exhaustive, 215  
 expansion by minors, 151  
**Explicit Arguments**, 229, 244, 245  
 explicit definition, 35  
 Explicit Definition, 113, 153, 154, 155, 156, 244, 245  
 EXPLICIT DEFINITION, 37  
 explicit result, 153  
 exponential, 88, 90, 137  
**Exponential**, 136  
 exponential form, 198  
 extend cyclically, 164, 174, 175, 178, 179  
 Extended, 103  
**Extended Precision**, 246  
 extra-lingual, 173

## —F—

**Factorial**, 171  
 factorial function, 47, 172  
*factorial function*, 91  
**Factors**, 235, 236  
 Falkoff, 261  
 falling factorial, 136, 172  
 family, 15  
*fcase*., 257  
**Fetch**, 188, 189  
 Fibonacci numbers, 241  
 figurate numbers, 50, 171  
**Files**, 268  
*fill*, 97, 149, 160, 167, 186  
 fit, 136, 141, 145, 146, 148, 160, 161, 162, 186, 199, 224, 279  
*fit*, 99, 167  
**Fit**, 172  
 fit conjunction, 91, 112  
**Fix**, 222  
 fixed seed, 209  
 floating point, 104, 105

**Floor**, 115, 116  
*for*., 155, 252  
**Foreign**, 173  
 foreign conjunction, 13, 21, 93  
*fork*, 9, 101, 119, 183  
 forks, 76, 77  
 FORKS, 11  
 formal name, 111  
 format, 133  
**Format**, 198, 199  
 FORMAT, 27  
 forward difference, 178  
 fractional arguments, 147  
*frame*, 95  
**fret**, 106, 164  
**From**, 184, 185  
*from function*, 52  
*function*, 7  
 function display, 173  
 Function table adverb, 32  
*function tables*, 7  
 FUNCTION TABLES, 67  
*functional*, 1

## —G—

gamma function, 171  
 Gaussian integer, 115  
**GCD**, 125, 128  
 general-purpose, 1  
 Geometric mean, 11  
 GEOMETRY, 81  
**gerund**, 45, 113, 138, 139, 164, 174, 175, 178, 179, 190, 200, 201, 203  
 global, 153  
 Global Parameters, 278  
 golden mean, 105  
*goto\_name*., 155, 253  
 grade, 75  
**Grade Down**, 180  
**Grade Up**, 177  
 grading, 210  
 grammar, 1, 19  
 GRAMMAR, 66  
 graphs, 71  
*greatest common divisor*, 125

## —H—

**Halve**, 132  
Hanoi, Tower of, 47  
**Head**, 186  
Hermitian matrix, 295  
Heron's formula, 26, 81  
**Hex Representation**, 271  
hierarchy, 1, 9, 10  
Hilbert matrix, 105  
hook, 76, 77, 119  
hook, 102  
Horner method, 46  
Horner's efficient evaluation, 45  
Host, 270  
host computer systems, 93  
host system, 173  
HOUSEKEEPING, 21  
Hui, 93, 200, 261  
hyperbolic, 231  
**Hypergeometric**, 223  
hypotenuse, 128, 129

## —I—

**Identity**, 182  
*identity element*, 58, 174  
*identity function*, 11, 174  
identity function, 213  
IDENTITY FUNCTIONS, 58  
identity matrix, 54, 134  
*if .*, 155, 254, 255  
imaginary, 128  
**Imaginary**, 125, 226  
imaginary number, 135  
immediate descendants, 83  
**Increment**, 120  
*in-degrees*, 83  
**Indeterminate**, 122  
**Index Of Last**, 225  
indices, 95  
*indirect*, 113  
indirect locative, 107  
indivisible part, 121, 130  
infinite, 204, 208  
infinite power, 23, 84  
infinity, 93, 121  
**Infinity**, 123

*infix*, 79  
**Infix**, 178  
*inflection*, 94  
informal name, 111  
inner product, 73, 145, 151  
input file, 173  
input prompt, 278  
*insert*, 200  
**Insert**, 174, 201  
insert adverb, 45  
Insertion on an empty list, 58  
**Integers**, 224  
integral, 86  
interval, 186  
intrinsic rank, 98  
inverse, 54, 56, 117, 129, 134, 137, 158, 170, 207  
*inverse*, 24  
Inverse adverb, 32  
invertible monads, 138  
iota, 224, 225  
*iota*, 111  
*isolated phrase*, 9  
*item*, 96, 131, 141, 224, 225, 252  
**Item Amend**, 190, 191  
itemize, 160, 162  
**Itemize**, 162  
ITERATION, 49  
Iverson, 261, 263

## —J—

**j**, 93  
J version, 279  
*joy*, 111  
JUNCTIONS, 78

## —K—

*k-cell*, 42  
k-cells, 97  
Kerner's Method, 90  
**Key**, 175, 176  
*key classification*, 79  
keyboard, 173, 267

## —L—

label\_name ., 155, 253  
**Laminate**, 162  
**Larger Of**, 119  
**Larger Than**, 118  
last item, 187, 193  
last occurrence, 225  
**LCM**, 128  
leading item, 186, 192  
leaf, 228  
least common multiple, 128  
**Left**, 181  
left-justified, 198  
**Length**, 128  
length of sides, 81  
Length of vector, 26  
Lengths of sides, 26  
**Less**, 131  
**Less Than**, 114  
**Less than or Equal**, 117  
**Lesser Of**, 115, 116  
Letter Frequency, 69  
**Lev**, 182  
level, 188, 238  
**Level**, 227, 228  
level conjunction, 156  
lexical, 96, 211  
limit, 138  
limited precision, 112  
line, 106  
linear, 13  
linear equations, 134  
LINEAR FUNCTIONS, 54  
linear representation, 275  
linearly independent, 134  
**Link**, 163  
*list*, 95  
literal, 95  
literal representation, 198  
local, 153  
**Local**, 113  
local name, 252  
locale, 107  
*locale*, 35  
locale, current, 294  
locale, named, 292  
locale, numbered, 292

**Locales**, 292  
*locative*, 35, 93, 107, 153  
locative, indirect, 107  
Locatives, 107  
**Logarithm**, 137  
logical *or*, 125  
lowest form, 125

## —M—

m ., 153, 229, 244, 245  
magnitude, 128, 237  
**Magnitude**, 147, 148  
main entry, 111  
major alphabet, 66  
**Map**, 188, 189  
match, 220  
**Match**, 132  
matrices, 1  
*matrix*, 95  
**Matrix Divide**, 134  
matrix product, 92, 151  
*matrix product*, 54  
Matrix product, 134  
**Max**, 119  
maximal cube, 164  
McDonnell, 119, 133, 261  
McIntyre, 261  
Mean, 77  
**Member of Interval**, 221  
memory limit, 279  
**Min**, 115, 116  
mine stope, 91  
minors, 151, 179  
**Minus**, 130  
mnemonic, 1  
MNEMONICS, 2  
mod, 147  
modulo, 147, 170  
**Monad**, 97  
**Monad / Dyad**, 157  
*monadic*, 97  
monadic function, 205  
monadic uses, 206  
*monomial*, 86  
*movement* vector, 164  
multinomial, 232  
multiplier, 232

multiplier and roots, 88

## —N—

*n.*, 153, 229, 244, 245

*Name*, 93

name class, 273

name list, 273

names, 222

Names, 31, 273

NAMES, 35

names changed, 273

nand, 129

*national use*, 93

**Natural Log**, 137

*natural logarithm*, 136

**Negate**, 130

negation, 129

negative indexing, 212

negative integers, 215

negative sign, 93, 198

**Negative Sign**, 121

*neutral*, 58, 174

Newton's Method, 89

*nodes*, 83

non-negative integers, 224

*non-overlapping*, 178, 179

non-singular, 134

*non-standard*, 215

nor, 126, 212

normalize, 90, 176

**Not**, 131

*not-and*, 212

**Not-And**, 129

notation, 93

**Not-Or**, 126

*noun*, 7

Noun, 94

Nouns, 67

NOUNS, 95

*nub*, 59

**Nub**, 145

**Nub Sieve**, 146

number of digits, 198

Number of divisors, 26

number of primes, 234

numbered, 293

**Numbers**, 197

NUMBERS, 66

numeric, 95, 186

**Numerical Functions**, 295

## —O—

*oblique*, 29, 79

**Oblique**, 175, 176

obverse, 138, 207, 213

**Obverse**, 158

OBVERSE, 56

**Odd**, 152

*odd function*, 38

omitted **T** block, 249, 254, 257, 260

one-rowed table:, 74

open, 95

**Open**, 118

*or*, 212

**Or**, 125

Or table, 68

*order of execution*, 9

out of, 8

**Out Of**, 171

*out-degrees*, 83

*outfix*, 79

**Outfix**, 179

output file, 173

over, 67

overlapping, 164

*overtake*, 186

*own*, 214

## —P—

*padding*, 97, 160

paren representation, 275

parentheses, 94, 202

parenthesis, 99

parenthesized, 203

parse table, 100, 286

PARSING, 99

*partial sums*, 29, 178

partitioning identities, 174

PARTITIONS, 29, 79, 80

parts of speech, 94

**PARTS OF SPEECH**, 67

**Passive**, 144

path, 188

**Path**, 293  
 period, 94  
 permanent, 151  
 permutation, 215  
 permutation functions, 52  
*permutation vector*, 52, 180  
 permutations, 211  
 PERMUTATIONS, 52  
**Permute**, 215, 216  
 pi, 237, 247  
**Pi Times**, 231  
**Plus**, 124  
**Polar**, 237  
 Polar coordinates, 128  
 polygons, 81  
 polyhedra, 81  
 Polynomial, 30, 45, 152, 232, 233  
 polynomial coefficients, 176  
 Polynomial difference, 86  
 Polynomial in terms of roots, 88  
 Polynomial product, 30  
 Polynomial sum, 86  
 polynomials, 134  
 POLYNOMIALS, 86, 87, 88, 89, 91  
 Positive numbers, 26  
 power, 233  
**Power**, 136, 138, 139, 140  
 POWER AND INVERSE, 23  
*power polynomial*, 91  
 Power table, 67  
 precedence, 83  
 prefix, 29, 79  
**Prefix**, 178  
*preparation*, 56  
 prime, 139, 203, 234  
**Prime**, 235, 236  
**Prime Exponents**, 235, 236  
**Prime Factors**, 235, 236  
 Prime select, 26  
 Prime test, 26  
 primes, 68  
**Primes**, 234  
 primitive, 2, 94  
 primitives, 1  
 print precision, 199, 278  
 probability, 131  
 procedure, 13  
 product scan, 168

products of numbers, 176  
 profile, 267  
 programming language, 1  
 PROGRAMS, 13  
 Progressive products, 178  
*projection*, 134  
*pronoun*, 7, 93  
 Pronoun, 94  
 Pronouns, 67  
**Properties of numbers**, 26  
 Propositions, 74  
 proverb, 143  
 Proverb, 67, 94  
 proxy, 142  
 punctuation, 155, 249  
 Punctuation, 67, 94  
 PUNCTUATION, 9

## —Q—

QR decomposition, 295  
*queue*, 100  
 quicksort, 254  
 quotes, 4, 93

## —R—

$\pi$ , 104  
 radians, 128, 231  
 random, 209  
 Random, 195  
 random permutation, 209  
*random seed*, 209, 278  
*rank*, 58, 95, 141  
 Rank, 42, 97, 194, 195  
 rank conjunction, 42, 248  
 rank-0 function, 217  
 ranks, 76, 95, 111, 204, 208, 213  
 Rational, 103  
 ravel, 141, 163, 212  
**Ravel**, 160  
**Ravel Items**, 161  
 raze, 220  
**Raze**, 163  
**Raze In**, 220  
 READING AND WRITING, 25  
 real, 128  
**Real**, 125

- reciprocal, 134
- Reciprocal**, 133
- reciprocals, 247
- RECURSION, 47
- recursively, 35, 222
- referent, 93, 113
- referents, 222
- reflection*, 134
- Reflexive**, 144
- Regular Expression, 291
- Reiter, 261
- relatively prime, 104
- remainder, 70, 147
- Remainder table, 26
- repeated squaring, 260
- repetitions, 167
- Representation, 274
- representations*, 89
- result, 155, 256
- result rank, 218
- return.*, 155, 256
- reversal, 224
- reverse, 164
- Reverse**, 149
- rhematic rules, 166
- Right**, 181
- right-justified, 198
- rising factorial, 223
- Roll**, 209
- Root**, 135
- root representation, 89
- roots, 88
- Roots**, 232, 233
- Roots from coefficients, 89
- roots of unity, 66
- run together, 150

## —S—

- Same**, 181
- SAMPLE TOPICS, 65
- scalar*, 95, 218
- screen, 173
- script, 153
- script index, 273
- Scripts, 106, 267
- Secant Slope**, 219
- security level, 279

- security violation, 279
- select.*, 155, 257
- Selection, 73
- self-classification, 145
- Self-classify**, 112
- self-inverse, 130
- self-reference, 35, 47
- Self-reference, 203
- Self-Reference**, 142
- semiperimeter, 81
- Semiperimeter, 26
- sequence of execution, 94, 154
- sets, 71
- Sets, 74
- shape, 42
- shape*, 95
- Shape**, 141
- Shape Of**, 141
- shard, 164, 178
- Shift**, 149
- signed* area, 81
- Signed area, 26
- signed volume, 26
- Signum**, 127
- silent, 267
- sin, 152
- sine, 231
- sine function, 134
- sinh, 152, 231
- skip test, 260
- Sockets, 290
- Sort**, 177, 180
- sorting, 210
- SORTING, 75
- Space, 277
- spaces, 186
- spanned, 134
- spelling, 1
- SPELLING, 65
- sphere, 134
- Spread**, 238
- square*, 136
- Square**, 129
- square root, 129
- square root*, 136
- Square Root**, 135
- stack*, 100
- standard cycle representation*, 215

standard error, 268  
 standard input, 268  
 standard output, 268  
 Std Deviation, 77  
 stderr, 268  
 stdin, 268  
 stdout, 268  
 Stegun, 261  
 Stirling numbers, 62, 92, 136  
**Stitch**, 161  
 stop, 267  
**Stop**, 281  
 stope, 136, 233  
 stope polynomials, 91, 136  
 Stopes, 91  
*structure of words*, 19  
*structured programming*, 13  
*subtotals*, 29, 171  
 Subtotals, 178  
 subtraction, 130  
 successive approximation, 89  
 successive axes, 184  
*suffix*, 79  
**Suffix**, 179  
 SUSPENSION, 109  
 SYMBOLIC FUNCTIONS, 82  
 symbol-less, 274  
 system type, 279

## —T—

**T** block, 249  
*table*, 95, 175  
**Table**, 174  
 Table of all permutations, 211  
 table of vertices, 81  
 table representation, 153  
*tacit*, 1  
 TACIT EQUIVALENTS, 40  
 tacit form, 153  
**Tail**, 187  
 tail end, 215  
**Take**, 186  
 tally, 42  
**Tally**, 167  
 tangent, 89, 231  
 tanh, 231  
 tautology, 12, 59, 267

**Taylor Approximation**, 243  
**Taylor Coefficient**, 240, 241  
 Taylor series, 88, 90  
**Taylor, Assign**, 239  
**Taylor, Weighted**, 242  
 tessellation, 164  
 three ranks, 194, 195  
**Tie**, 200  
*tie conjunction*, 45  
**Time**, 276  
 tolerance, 99, 104, 112, 114, 127, 132, 146,  
     148, 172, 224, 279  
 tolerantly equal, 145  
 totient function, 236  
 Tower of Hanoi, 47  
 trace, 285  
 train, 203  
*Train*, 201  
 trains, 76, 183  
*trains*, 99  
 TRAINS, 101  
 transpose, 152  
 tree, 13, 83  
 Tree displays, 102  
 tree representation, 275  
*triangular numbers*, 171, 250  
*trident*, 9, 102  
 try., 155, 259

## —U—

u., 153, 244, 245  
*unbounded*, 97  
 under, 115, 119, 126  
*under*, 206  
**Under**, 207  
 UNDER, 56  
 underbar, 3, 93  
 unit axis, 162  
 unit circle, 134, 237  
*utilities*, 7

## —V—

v., 153, 244, 245  
 valence, 4, 98  
 variant, 233  
 variant of the power, 91



*variants*, 172  
*vector*, 95  
vector calculus, 218  
vectors, 1  
verb, 1  
Verb, 94  
*verb definition*, 13  
VERBS, 97  
VERBS AND ADVERBS, 7  
Verbs/Proverbs, 67  
VOCABULARY, 19, 311  
*volume*, 81

## —W—

weighted sum, 168  
*while* ., 155, 260  
*whilst* ., 155, 260  
width allocated, 198  
**Window Driver**, 280  
*with*, 98  
within a line, 286  
*without repetition*, 209  
word formation, 114  
**Word Formation**, 166  
WORD FORMATION, 33  
*write*, 19

## —X—

x, 103  
x ., 153, 244, 245

## —Y—

y ., 153, 244, 245

## —Z—

*zero items*, 174  
zeros, 89  
*zeros*, 88

# Table of Contents

<b>INTRODUCTION.....</b>	<b>3</b>
1. MNEMONICS .....	4
2. AMBIVALENCE.....	6
3. VERBS AND ADVERBS .....	9
4. PUNCTUATION .....	11
5. FORKS .....	13
6. PROGRAMS .....	16
7. BOND CONJUNCTION.....	18
8. ATOP CONJUNCTION .....	21
9. VOCABULARY.....	23
10. HOUSEKEEPING .....	26
11. POWER AND INVERSE .....	28
12. READING AND WRITING .....	30
13. FORMAT .....	32
14. PARTITIONS .....	34
15. DEFINED ADVERBS.....	36
16. WORD FORMATION .....	38
17. NAMES AND DISPLAYS.....	40
18. EXPLICIT DEFINITION .....	42
19. TACIT EQUIVALENTS.....	45
20. RANK .....	47
21. GERUND AND AGENDA .....	50
22. RECURSION.....	53
23. ITERATION.....	55
24. TRAINS.....	57
25. PERMUTATIONS.....	58
26. LINEAR FUNCTIONS.....	60
27. OBVERSE AND UNDER .....	62
28. IDENTITY FUNCTIONS AND NEUTRALS.....	64
29. SECONDARIES .....	68
<b>SAMPLE TOPICS.....</b>	<b>71</b>
1. SPELLING .....	71
2. ALPHABET AND NUMBERS .....	72
3. GRAMMAR.....	72
4. FUNCTION TABLES.....	73
5. BORDERING A TABLE .....	74
6. TABLES (LETTER FREQUENCY) .....	75
7. TABLES .....	76
8. CLASSIFICATION .....	77
9. DISJOINT CLASSIFICATION (GRAPHS) .....	78

10. CLASSIFICATION (WITH SELECTION AND INNER PRODUCT).....	79
11. CLASSIFICATION (SETS AND PROPOSITIONS).....	80
12. SORTING .....	81
13. COMPOSITIONS (BASED ON CONJUNCTIONS) .....	82
14. COMPOSITIONS (BASED ON HOOKS AND FORKS) .....	83
15. JUNCTIONS .....	84
16. PARTITIONS (ADVERBS) .....	85
17. PARTITIONS (BASED ON CUT CONJUNCTION).....	86
18. GEOMETRY .....	87
19. SYMBOLIC FUNCTIONS.....	88
20. DIRECTED GRAPHS .....	89
21. CLOSURE.....	90
22. DISTANCE .....	91
23. POLYNOMIALS .....	92
24. POLYNOMIALS (CONTINUED).....	93
25. POLYNOMIALS IN TERMS OF ROOTS .....	94
26. POLYNOMIALS:     ROOTS FROM COEFFICIENTS (NEWTON’S METHOD).....	95
27. POLYNOMIALS:     ROOTS FROM COEFFICIENTS (KERNER’S METHOD).....	96
28. POLYNOMIALS:     STOPES .....	97
<b>DICTIONARY.....</b>	<b>99</b>
I: ALPHABET AND WORDS.....	99
II. GRAMMAR.....	101
A. <i>Nouns</i> .....	102
B. <i>Verbs</i> .....	105
C. <i>Adverbs and Conjunctions</i> .....	107
D. <i>Comparatives</i> .....	108
E. <i>Parsing and Execution</i> .....	108
F. <i>Trains</i> .....	110
G. <i>Extended and Rational Arithmetic</i> .....	112
H. <i>Frets and Scripts</i> .....	115
I. <i>Locatives</i> .....	116
J. <i>Errors and Suspension</i> .....	118
III. DEFINITIONS .....	120
<b>REFERENCES .....</b>	<b>275</b>
<b>ACKNOWLEDGMENTS .....</b>	<b>277</b>
<b>APPENDIX .....</b>	<b>279</b>
<i>Scripts</i> 0!: .....	281
<i>Files</i> 1!: .....	282
<i>Host</i> 2!: .....	285

<i>Conversions</i>	3!:	287
<i>Names</i>	4!:	288
<i>Representation</i>	5!:	289
<i>Time</i>	6!:	291
<i>Space</i>	7!:	292
<i>Global Parameters</i>	9!:	293
<i>Window Driver</i>	11!:	295
<i>Debug</i>	13!:	296
<i>Data Driver</i>	14!:	303
<i>Dynamic Link Library</i>	15!:	304
<i>Sockets</i>	16!:	305
<i>Regular Expression</i>	17!:	306
<i>Locales</i>	18!:	307
<i>Numerical Functions</i>	128!:	310
<b>INDEX</b>		<b>311</b>

= Self-Classify • Equal	=. Is (Local)	=: Is (Global)	121
< Box • Less Than	<. Floor • Lesser Of (Min)	<: Decrement • Less Or Equal	123
> Open • Larger Than	>. Ceiling •Larger Of (Max)	>: Increment • Larger Or Equal	127
_ Negative Sign /Infinity	_. Indeterminate	_: Infinity	130
+ Conjugate • Plus	+. Real/Imag • GCD (Or)	+: Double • Not-Or	133
* Signum • Times	*. Length/Angle•LCM (And)	*: Square • Not-And	136
- Negate • Minus	-. Not • Less	-: Halve • Match	140
% Reciprocal • Divided By	%. Matrix Inverse • Mat Div	%; Square Root • Root	143
^ Exponential • Power	^. Natural Log • Logarithm	^: <b>Power</b>	146
\$ Shape Of • Shape		\$: Self-Reference	151
~ <b>Reflex • Pass • EVOKE</b>	~. Nub •	~: Nub Sieve • Not-Equal	142
Magnitude • Residue	. Reverse • Rotate (Shift)	: Transpose	157
. <b>Det • Dot Product</b>	.. <b>Even</b>	.: <b>Odd</b>	161
: <b>Explicit •</b>	:. <b>Obverse</b>	:: <b>Adverse</b>	152
<b>Monad/Dyad</b>			
, Ravel • Append	,. Ravel Items • Stitch	,: Itemize • Laminate	173
; Raze • Link	;. <b>Cut</b>	;: Word Formation •	176
# Tally • Copy	#. Base 2 • Base	#: Antibase 2 • Antibase	180
! Factorial • Out Of	!. <b>Fit (Customize)</b>	!: <b>Foreign</b>	184
/ <b>Insert • Table</b>	/. <b>Oblique • Key</b>	/: Grade Up • Sort	187
\ <b>Prefix • Infix</b>	\. <b>Suffix • Outfix</b>	\: Grade Down • Sort	191
[ Same • Left	[. <b>Lev</b>	[ : Cap	194
] Same • Right	] . <b>Dex</b>	] : <b>Identity</b>	194
{ Catalogue • From	{. Head • Take	{ : Tail • { :: Map • Fetch	197
} <b>Item Amend • Amend</b>	} . Behead • Drop	} : Curtail •	203
" <b>Rank</b>	". Do • Numbers	" : Default Format • Format	207
~ <b>Tie (Gerund)</b>		~ : <b>Evoke Gerund</b>	213
@ <b>Atop</b>	@. <b>Agenda</b>	@: <b>At</b>	215
& <b>Bond /Compose</b>	&. <b>Under (Dual)</b>	&: <b>Appose</b>	218
? Roll • Deal	?. Roll • Deal (fixed seed)		222
a. <i>Alphabet</i>	a: <i>Ace</i> (boxed empty)	A. Anagram Index • Anagram	223
b. <i>Boolean/Basic</i>	c. Characteristic Values	C. Cycle-Direct • Permute	225
d. <b>Derivative</b>	D. <b>Derivative</b>	D: <b>Secant Slope</b>	230
e. Raze In • Member	E. • Member of Interval	f. <b>Fix</b>	234
H. <b>Hypergeometric</b>	i. Integer • Index Of	i: • Index Of Last	237
j. Imaginary • Complex	L. <b>L: Level, Level</b>	m. n. Explicit Noun Args	240
NB. Comment	o. Pi Times • Circle Fn	p. Polynomial	244
p: Prime	q: Prime Factors • Prime	r. Angle • Polar	248
	Exp		
s: <b>Spread</b>	t. <i>Taylor Coefficient</i>	t: <i>Weighted Taylor</i>	252

<b>T. Taylor Approx</b>	<b>u. v.</b> Explicit Verb Args	<b>x. y.</b> Explicit Arguments	257
<b>x:</b> Extended Precision	<b>_9:</b> to <b>9:</b> Constant Fns	Control Structures	260

**VOCABULARY**