

Shakti tutorial

John Estrada

21 December 2020

Copyright © 2020 John Estrada

1 Introduction

The k9 programming language is designed primarily for the analysis of data. It may surprise new users with two rather different paradigms, (1) fast data analysis and (2) concise syntax. After you become familiar with the languages, these features will both seem normal and going back to slow and verbose programming will be surprisingly difficult.

1.1 Going fast

Imagine you have a small, on-disk, 100 million row database containing a time-series with two float values at each time. Additionally this data could be split in three different tables covering different measurements. Here's how fast k9 can read in the data from disk and compute a statistic, average difference over each table, which uses each and every row.

```
bash-3.2$ 2020.xx.xx 17GB 4core (c) shakti
\t q:2:`q;{select s:avg a-b from x}`q[!q]
495
```

That's 495 ms to read the data in from disk and compute over all the 100 million values. The data read is the biggest bit. If the data were already in memory then the computation would be faster:

```
\t {select s:avg a-b from x}`q[!q]
230
```

230 ms, not bad for 100 million calculations.

The code to generate the on-disk database is presented below. Speed of course will depend on your hardware so times will vary.

```
nf:d+*|d:(|-d),d:683 954 997 1000;
T:~`t ?[;_8.64e7]@
B:100++\1e-2*-3+nf bin/:?[*|nf]@
S:?[;1e-2*2,2,8#1]@
L:{select t,b,a:b+s from +`t`b`s!(T;B;S)@`x}
q:`eurusd`usdjpy`usdchf!L'60e6 20e6 20e6
`q 2:q
```

1.2 Going concise

The k9 language is more closely related to mathematics syntax than most programming languages. It requires the developer to learn to “speak” k9 but once that happens most find an ability to speak quicker and more accurately in k9 than in other languages. At this point an example might help.

In mathematics, “3+2” is read as “3 plus 2” as you learn at an early age that “+” is the “plus” sign. For trivial operations like arithmetic most programming languages use symbols also. When moving beyond arithmetic, most programming languages switch to words while k9 remains with symbols. As an example, to determine the distinct values of a list most programming languages might use a syntax like `distinct()` while k9 uses `?`. This requires the developer to learn how to say a number of symbols but once that happens it results in much shorter code that is quicker to write, easier to inspect, and easier to maintain.

This should not be surprising. In arithmetic, which do you find easier to answer?

Math with text

Three plus two times open parenthesis six plus fourteen close parenthesis

Math with symbols

$3+2*(6+14)$

In code, which will you eventually find easier to understand?

Code with text

```
x = (0,12,3,4,1,17,-5,0,3,11);y=5;
distinct_x = distinct(x);
gt_distinct_x = [i for i in j if i >= y];
```

Code with symbols

```
x:(0,12,3,4,1,17,-5,0,3,11);y:5;
z@&y<z:?x
```

If you're new to k9 then you'll appreciate that symbols are shorter but look like line noise. That's true but so did arithmetic until you learned the basics.

When you first learned arithmetic you likely didn't have a choice. Now you have a choice whether or not you want to learn k9. If you give it a try, then you'll get it quickly and move onto the power phase fast enough that you'll be happy you gave it a chance.

1.3 Get k9.

`https://shakti.sh`

You will find the Linux version in the linux directory and the MacOS version under macos. Once you download the MacOS version you'll have to change its file permissions to allow it to execute.

```
chmod u+x k
```

Again on the mac if you then attempt to run this file you likely won't succeed due to MacOS security. You'll need to go to "System Preferences..." and then "Security and Privacy" and select to allow this binary to run. (You'll have to have tried and failed to have it appear here automatically.)

1.4 Help/Info Card

Typing \ in a terminal window gives you a concise overview of the language. This document aims to provide details to beginning user where the help screen is a bit too terse. Some commands are not yet complete and thus marked with an asterisk, eg. *update A by B from T where C.

Verb		Adverb		Type		System
:	x	y	f/ over c/	+join	char " ab"	\l a.k
+	flip	plus	f\ scan c\	+split	name ``ab	\t:n x
-	negate	minus	f' each v'	+has	int 0 2 3	\u:n x

```

*   first      times      f': eachp v': +bin   flt  0 2 3.      \v
%   divide     f/: eachr ([n;]f)/:  date 2001.01.01   .z.d
&   where      min/and    f\': eachl ([n;]f)\:  time 12:34:56.789 .z.t
|   reverse    max/or
<   asc        less
>   dsc        more
=   group      equal      I/O
~   not        match      0:  r/w line          Class          \f
!   enum       key        1:  r/w byte          List (2;3.4;`c)    \ft x
,   enlist     cat        *2: r/w data          Dict [a:2;b:`c]    \fl x
^   sort       [f]cut      *3: k-ipc set          Func {[a;b]a+b}    \fc x
#   count      [f]take     *4: https get          Expr :a+b          \cd [d]
_   floor      [f]drop
$   string     cast       $[b;t;f] Control    Table
?   unique+    find+      t:[[]i:2 3;f:3 4.;s:`a`b]
@   type       [f]at       @[r;i;f[;y]] amend  utable [[b:..]a:..]
.   value      [f]dot      .[r;i;f[;y]] dmend  xtable `...![[a:..]
\\  exit       / comment   [if do while]

```

```

select[C]{count first last sum min max *[avg var dev med ...]}from T where B
update C from T where B; delete from T where B; key unkey meta in bin within
exp log sin cos sqr sqrt div mod bar; top freq msum mavg sums deltas prm cmb

```

1.5 rlwrap

Although you only need the `k` binary to run `k9` most will also install `rlwrap`, if not already installed, in order to get command history in a terminal window. `rlwrap` is “Readline wrapper: adds readline support to tools that lack it” and allows one to arrow up to go through the command buffer history.

In order to start `k9` you should either run `k` or `rlwrap k` to get started. Here I will show both options but one should run as desired. In this document lines with input are shown with a leading space and output will be without. In the examples below the user starts a terminal window in the directory with the `k` binary file. Then the users enters `rlwrap ./k RET`. `k9` starts and displays the date of the build, (c), and shakti and then listens to user input. In this example I have entered the command to exit `k9`, `\\`. Then I start `k9` again without `rlwrap` and again exit the session.

```

rlwrap ./k
Sep 13 2020 16GB (c) shakti
\\

./k
Sep 13 2020 16GB (c) shakti
\\
```

1.6 Simple example

JOHN: I SUGGEST MOVING THIS TO LATER BECAUSE AS IT IS, IT WOULDN'T LOOK SIMPLE AT ALL TO A NEWBIE.

Here I will start up `k9`, perform some trivial calculations, and then close the session. After this example it will be assumed the user will have a `k9` session running and working in repl mode. Comments (`/`) will be added to the end of lines as needed.

```

rlwrap ./k
Sep 13 2020 16GB (c) shakti
n:10000 / n data points
s:`a`b`c / data for symbols a, b, and c
q:+s!(-1+n?3;-1+n?3;-1+n?3) / table of returns (-1,0,1) for each symbol
q / print out the table
a b c
-- -- --
0 1 1
-1 -1 0
-1 1 1
0 1 -1
-1 -1 -1
..
```

At this point you might want to check which symbol has the highest return, most variance, or any other analysis on the data.

```

#'=+(+q) [] / count each unique a/b/c combination
a b c |
```

```

-- -- --|---
 0  1  1|407
-1 -1 -1|379
-1  0  0|367
 0 -1 -1|391
 1  1  1|349
..
+-1#+\q                                / calculate the return of each symbol
a|-68
b|117
c|73
{(+/m*m:x-avg x)%#x}' +q / calculate the variance of each symbol
a|0.6601538
b|0.6629631
c|0.6708467

```

Now let's exit the session.

```

\\
bash-3.2$

```

1.7 Document formatting for code examples

This document uses a number of examples to familiarize the reader with k9. The syntax is to have input with a leading space and output without a leading space. This follows the terminal syntax where the REPL input has space but prints output without.

```

3+2 / this is input
5   / this is output

```

1.8 k9 Idiosyncracies

One will need to understand some basic rules of k9 in order to progress. These may seem strange at first but the faster you learn them, the faster you'll move forward. Also, some of them, like overloading based on number of arguments, add a lot of expressability to the language.

1.8.1 The language changes often (for now).

There may be examples in this document which work on the version indicated but do not with the version currently available to download. If so, then feel free to drop the author a note. Items which currently cause an error but are likely to come back 'soon' will be left in the document.

1.8.2 Colon (:) is used to set a variable to a value

`a:3` is used to set the variable, `a`, to the value, `3`. `a=3` is an equality test to determine if `a` is equal to `3`.

1.8.3 Percent (%) is used to divide numbers

Yeah, `2` divided by `5` is written as `2%5`, not `2/5`.

1.8.4 Evaluation is done right to left

$2+5*3$ is 17 and $2*5+3$ is 16. $2+5*3$ is first evaluated on the right most portion, $5*3$, and once that is computed then it proceeds with $2+15$. $2*5+3$ goes to $2*8$ which becomes 16.

1.8.5 There is no arithmetic order

$+$ has equal precedence as $*$. The order of evaluation is done right to left unless parenthesis are used. $(2+5)*3 = 21$ as the $2+5$ in parenthesis is done before being multiplied by 3.

1.8.6 Operators are overloaded depending on the number of arguments.

```
*(13;6;9)    / single argument: * returns the first element
13
2*(13;6;9)   / two arguments: * is multiplication
26 12 18
```

1.8.7 Lists and functions are very similar.

k9 syntax encourages you to treat lists and functions in a similar function. They should both be thought of a mapping from a value to another value or from a domain to a range. Lists and functions do not have the same type.

```
1:3 4 7 12
f:{3+x*x}
1@2
7
f@2
7
@1
`I
@f
`.`
```

1.8.8 k9 notions of Noun, Verb, and Adverb

k9 uses an analogy with grammar to describe language syntax. The k9 grammar consists of nouns (data), verbs (functions) and adverbs (function modifiers).

- The boy ate an apple. (Noun verb noun)
- The girl ate each olive. (Noun verb adverb noun)

In k9 as the Help/Info card shows data are nouns, functions/lists are verbs and modifiers are adverbs.

- $3 > 2$ (Noun verb noun)
- $3 >' (1\ 12;1\ 4\ 5)$ (Noun verb adverb noun)

2 Examples

[JOHN: THIS SHOULD GO AFTER YOU EXPLAIN THE LANGUAGE. AGAIN, IT'S TOO SCARY AS OF NOW.] Before jumping into syntax, let's look at some example problems to get a sense of the speed of k9 at processing data. Given the author's background many of the examples will be based on financial markets. For those not familiar with this field a short introduction may help.

2.1 A Tiny Introduction to Financial Market Data

Financial market data generally are stored as prices (often call quotes) and trades. At a minimum, prices will include time (t), price to buy (b for bid) and price to sell (a for ask). Trades will include at a minimum time (t) and trade price (p). In normal markets there are many more prices than trades. (Additionally, the data normally includes the name of the security (s) and the exchange (x) from which the data comes.)

Let's use k9 to generate a set of random prices.

```
n:10
T:~10:00+`t n?36e5
B:100++\~1+n?3
A:B+1+n?2
q:+`t`b`a!(T;B;A);q
t          b    a
-----
10:01:48.464 100 102
10:23:12.033 100 102
10:30:00.432 101 102
10:34:00.383 101 103
10:34:36.839 101 102
10:42:59.230 100 102
10:46:50.478 100 102
10:52:42.189  99 100
10:55:52.208  99 101
10:59:06.262  98  99
```

Here you see that at 10:42:59.230 the prices update to 100 and 102. The price one could sell is 100 and the price to buy is 102. You might think that 100 seems a bit high so sell there. Later at 10:59:06.262 you might have thought the prices look low and then buy at 99. Here's the trade table for those two transactions.

```
t:+`t`p!(10:43:00.230 10:59:07.262;;100 99);t
t          p
-----
10:43:00.230 100
10:59:07.262  99
```

You'll note that the times didn't line up, because it apparently took you a second to decide to trade. Because of this delay, you'll often have to look back at the previous prices to join trade (t) and quote (q) data.

Now that you've learned enough finance to understand the data, let's scale up to larger problems to see the power of k9.

2.2 Data Manipulation

Generate a table of random data and compute basic statistics quickly. The data here includes time (t), security (s), and price delta (d). This table takes about 4 GB and 3.3 seconds on a relatively new consumer laptop.

```
n:_100e6 / 100 million rows
t:{09:00:00.000+x?10:00:00.000} / random times
s:{x?`a`b`c`d`e} / random symbols
m:0,(|m),365378984,m:271810244 42800467 2636454 62769 572 2;
d:{(-6+!13)@(+\m)bin x?_1e9}
\t q:+`t`s`d!(t[n];s[n];d[n]) / time data generation in ms
3391
```

As this point one might want to check start and stop times, see if the symbol distribution is actually random and look at the distribution of the price deltas.

```
select ti:min t, tf:max t from q / min and max time values
ti|09:00:00.000
tf|18:59:59.999
```

```
select c:#s by s from q / count each symbol
s|c
-|-----
a|20003490
b|19997344
c|19998874
d|20000640
e|19999652
```

```
select c:#d by d from q / check the normal distribution (2s to run)
d |c
--|-----
-6|1
-5|55
-4|6226
-3|263801
-2|4280721
-1|27179734
0|36531595
1|27188092
2|4279872
3|263610
4|6245
5|48
```

```
select gain:sum d by s from q / profit (or loss) over each symbol
```

```

s|gain
-|-----
a| 872
b| 2765
c| 2668
d| 2171
e|-2354

select loss:min +\d by s from q / worst loss over the period
s|loss
-|-----
a|-1803
b| -846
c|-2732
d|-2101
e|-2903

```

2.3 Understanding Code Examples

In the shakti mailing list there are a number of code examples that can be used to learn best practices. In order to make sense of other people's codes, one needs to be able to efficiently understand k9 language expressions. Here is an example of how one goes about this process.

```

ss:{*{
  o:o@&(-1+(#y)+*x@1)<o:1_x@1;
  $[0<#x@1;((x@0),*x@1;o);x]}[;y]/:(();&(x@(!#x)+\!#y)~\y)
}
```

This function finds a substring in a string.

```

00000000001111111112222222222333333
012345678901234567890123456789012345
"Find the +++ needle in + the ++ text"
```

Here one would expect to find “++” at 9 and 29.

```

ss["Find the +++ needle in + the ++ text";"++"]
9 29
```

In order to determine how this function works let's strip out the details...

```

ss:{
  *{
    o:o@&(-1+(#y)+*x@1)<o:1_x@1; / set o
    $[0<#x@1;((x@0),*x@1;o);x] / if x then y else z
  }
  [;y]/:(();&(x@(!#x)+\!#y)~\y) / use value for inner function
}
```

Given that k9 evaluates right to left, let's start with the rightmost code fragment.

```

(();&(x@(!#x)+\!#y)~\y) / a list (null;value)
```

And now let's focus on the value in the list.

```
&(x@(!#x)+\!#y)~\y
```

In order to easily check our understanding, we can wrap this in a function and call the function with the parameters shown above. In order to step through, we can start with the inner parenthesis and build up the code until it is complete.

```
{!#x}["Find the +++ needle in + the ++ text";"++"]
{!#x}["Find the +++ needle in + the ++ text";"++"]
~
:rank
```

This won't work as one cannot call a function with two arguments and then use only one. In order to get around this, we will insert code for the second argument but not use it.

```
{y;#x}["Find the +++ needle in + the ++ text";"++"]
36
{y;!#x}["Find the +++ needle in + the ++ text";"++"]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 ..
```

As might have been guessed `#x` counts the number of characters in the first argument and then `!#x` generates a list of integers from 0 to `n-1`.

```
{(!#x)+\!#y}["Find the +++ needle in + the ++ text";"++"]
0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
10 11
11 12
12 13
13 14
14 15
15 16
16 17
17 18
18 19
19 20
20 21
..
```

Here the code takes each integer from the previous calculation and then adds an integer list as long as the second argument to each value. In order to verify this, you could write something similar and ensure the output what you predicted.

```
{(!x)+\!y}[6;4]
```


3 Benchmark

[JOHN: THIS TOO SHOULD GO AFTER EXAMPLES BUT AFTER YOU EXPLAIN THE LANGUAGE] Shakti is a fast data analysis language and clear benchmarks illustrate this. The Shakti website has a file for such purpose, b.k. For example, you can see below that the first query (Q1) k9 takes 1ms while postgres, spark and mongo are orders of magnitude slower.

```

b.k
T:{09:30:00+_6.5*3600*(!x)%x}
P:{10+x?90};Z:{1000+x?9000};E:?[;"ABCD"]

/m:2;n:6
m:7000;n:5600000;
S:(-m)?^4;N:|1+_n*{x%+/x:exp 15*(!x)%x}m

t:S!{+^t`e`p`z!(T;E;P;Z)@'x}'N
q:S!{+^t`e`b!(T;E;P)@'x}'6*N

a:*A:100#S

\t {select max p by e from x}'t A
\t {select sum z by `o t from x}'t A
\t:10 {select last b from x}'q A
\t:10 select from t[a],`t`q a where p<b
\

C:M:?[;"ABCDEFGHJIJ"]
trade(sym time exchange price size cond)
quote(sym time exchange bid bz ask az mode)

          Q1      Q2      Q3      Q4  ETL   RAM   DSK
k          1        9        9        1
postg      71000    1500    1900    INF  200    1.5    4.0
spark     340000    7400    8400    INF  160   50.0    2.4
mongo      89000    1700    5800    INF  900    9.0   10.0

960 billion quotes (S has 170 billion. QQQ has 6 billion.)
48 billion trades (S has 12 billion. QQQ has 80 million.)

```

3.1 Understanding the benchmark script

3.1.1 T

T is a function which generates a uniform list of times from 09:30 to 16:00.

```

T:{09:30:00+_6.5*3600*(!x)%x}
T[13]                / 13 times with equal timesteps over [start;end)
~09:30:00 10:00:00 10:30:00 11:00:00 11:30:00 .. 15:00:00 15:30:00

```

```
?1_-' :T[10000] / determine the unique timesteps
?00:00:02 00:00:03
```

3.1.2 P, Z, E

P is a function to generate values from 10 to 100 (price). Z is a function to generate values from 100 to 1000 (size). E is a function to generate values A, B, C, or D (exchange).

```
P[10]
78 37 56 85 40 68 88 50 41 78
Z[10]
4820 2926 1117 4700 9872 3274 6503 6123 9451 2234
E[10]
"AADCBCCBC"
```

3.1.3 m, n, S, N

m is the number of symbols. n is the number of trades. S is a list of symbol names. N is a list of numbers in decreasing order which sum approximately to n.

```
4#S
`EEFD`IOHJ`MEJO`DHNK
4#N
11988 11962 11936 11911
+ /N
5604390
```

3.1.4 t

t is an xtable of trades. The fields are time (t), exchange (e), price (p), and size (z). The number of trades is set by n.

Pulling one random table from t and showing 10 random rows.

```
10?*t@1?S
t          e p  z
----- - -- ----
14:37:53 D 73 4397
11:43:25 B 20 2070
10:21:18 A 53 6190
13:26:03 C 33 7446
14:07:06 B 13 2209
15:08:41 D 12 4779
14:27:37 A 11 6432
11:22:53 D 92 9965
11:12:37 A 14 5255
12:24:28 A 48 3634
```

3.1.5 q

q is a xtable of quotes. The fields are time (t), exchange (e), and bid (b). The number of quotes is set to 6*n.

```
10?*q@1?S
```

```

t          e b
----- - --
11:31:12 A 80
14:08:40 C 63
14:05:07 D 12
11:31:43 A 56
12:44:19 A 45
10:13:21 A 71
15:19:08 A 74
13:42:20 D 43
11:31:41 D 66
14:41:38 A 63

```

3.1.6 a, A

a is the first symbol of S. A consists of the first 100 symbols of S.

```

a
`PKEM

```

3.1.7 Max price by exchange

The query takes 100 tables from the trade xtable and computes the max price by exchange.

```

*{select max p by e from x}'t A
e|p
-|--
A|99
B|99
C|99
D|99
\t {select max p by e from x}'t A
22

```

3.1.8 Compute sum of trade size by hour.

This query takes 100 tables from the trade xtable and computes the sum of trade size done by hour.

```

*{select sum z by `o t from x}'t A
t |z
--|-----
09| 4885972
10|10178053
11|10255045
12|10243846
13|10071057
14|10203428
15|10176102
\t {select sum z by `o t from x}'t A
27

```


3.1.9 Compute last bid by symbol

This query takes the 100 tables from the quote xtable and returns the last bid.

```
3?{select last b from x}'q A
b
--
18
98
85

\t:10 {select last b from x}'q A
2
```

3.1.10 Find trades below the bid

This query operates on one symbol from the q and t xtables, i.e. a single quote and trade table. The quote table is joined to the trade table giving the current bid on each trade.

```
4?select from t[a],`t`q a where p<b
t          e p z      b
----- - -- ---- --
13:54:35 B 94 1345 96
11:59:52 C 26 1917 89
10:00:44 C 40 9046 81
10:59:39 A 25 5591 72
\t:10 select from t[a],`t`q a where p<b
3
```

4 Verbs

This chapter covers verbs which are the core primitives of k9. Given how different is it to call functions in k9 than in most other languages, you may have to read this chapter a few times. Once you can “speak” k9 you’ll read `|x` better than `reverse(x)`.

Most functions are overloaded and change depending on the number of arguments. This can confuse new users. Eg. `(1 4)++(2 3;5 6;7 8)` contains the plus symbol once as flip and then for addition. (Remember evaluation is right to left!)

When functions are overloaded into materially different uses, I flag this with a '+' suffix. For example, `unique+` and `find+` differ depending on whether their argument is a scalar or a vector.

There are also verbs that can take a function argument. I flag those with a '[f]' prefix. These include `[f]cut`, `[f]take`, `[f]drop`, `[f]at`, and `[f]dot`.

```
Verb
: [x], page 16,      [set], page 16.
+ [flip], page 17,   [plus], page 17.
- [negate], page 18, [minus], page 18.
* [first], page 18,  [times], page 18.
%           [divide], page 19.
& [where], page 19,  [min/and], page 19.
| [reverse], page 19, [max/or], page 20.
< [asc], page 20,     [less], page 20.
> [asc], page 20,     [less], page 20.
= [group], page 21,   [equal], page 21.
~ [not], page 21,     [match], page 22.
! [enum], page 22,    [key], page 23.
, [enlist], page 23,  [cat], page 23.
^ [sort], page 24,    [[f]cut], page 24.
# [count], page 25,   [[f]take], page 25.
_ [floor], page 26,   [[f]drop], page 26.
$ [string], page 27,  [cast], page 27.
? [unique+], page 27, [find+], page 27.
@ [type], page 28,    [[f]at], page 29.
. [value], page 29,   [[f]dot], page 30.
```

4.1 `x ⇒ :x`

4.2 `set ⇒ x:y`

Set a variable, `x`, to a value, `y`. There should be no space between `x` and `:`.

```
a:3
a
3
b:(`green;37;"blue")
b
```

```

green
37
blue
c:{x+y}
c
{x+y}
c[12;15]
27

```

4.3 flip \Rightarrow +x

Flip, or transpose, x.

```

x:((1 2);(3 4);(5 6))
x
1 2
3 4
5 6
+x
1 3 5
2 4 6
`a`b!+x
a|1 3 5
b|2 4 6
+`a`b!+x
a b
- -
1 2
3 4
5 6

```

4.4 plus \Rightarrow x+y

Add x and y. Arguments can be elements or lists but if both x and y are lists then they must be of equal length.

```

3+7
10

a:3;
a+8
11

3+4 5 6 7
7 8 9 10

3 4 5+4 5 6
7 9 11

```

```

3 4+1 2 3      / lengths don't match
:length

```

```

10:00+1      / add a minute
10:01

```

```

10:00:00+1    / add a second
10:00:01

```

```

10:00:00.000+1 / add a millisecond
10:00:00.001

```

4.5 negate \Rightarrow -x.

```

-3
-3
--3
3
x:4;
-x
-4
d:`a`b!((1 2 3);(4 5 6))
-d
a|-1 -2 -3
b|-4 -5 -6

```

4.6 minus \Rightarrow x-y.

Subtract y from x.

```

5-2
3
x:4;y:1;
x-y
3

```

4.7 first \Rightarrow *x

Return the first value of x.

```

*1 2 3
1
*((1 2);(3 4);(5 6))
1 2
**((1 2);(3 4);(5 6))
1
*`a`b!((1 2 3);(4 5 6))
1 2 3
*|1 2 3
3

```

4.8 times \Rightarrow x*y

Mutliply x and y.

```
3*4
12
3*4 5 6
12 15 18
1 2 3*4 5 6
4 10 18
```

4.9 divide \Rightarrow x%y

Divide x by y.

```
12%5
2.4
6%2    / division of two integers returns a float
3f
```

4.10 where \Rightarrow &x

Given a list of boolean values return the indices where the value is non-zero.

```
&001001b
2 5
"banana"="a"
010101b
&"banana"="a"
1 3 5
x@&30<x:12.7 0.1 35.6 -12.1 101.101 / return values greater than 30
35.6 101.101
```

Given a list of non-negative integer values, eg. x[0], x[1], ..., x[n-1], generate x[0] values of 0, x[1] values of 1, ..., and x[n-1] values of n-1.

```
& 3 1 0 2
0 0 0 1 3 3
```

4.11 and \Rightarrow x&y

The smaller of x and y. One can use the over adverb to determine the min value in a list.

```
3&2
2
1 2 3&4 5 6
1 2 3
010010b&111000b
010000
`a&`b
`a
&/ 3 2 10 -200 47
-200
```

4.12 reverse \Rightarrow |x

Reverse the list x.

```
|0 3 1 2
2 1 3 0
|"banana"
"ananab"
|((1 2 3);4;(5 6))
5 6
4
1 2 3
```

To get the last element, one can take the first element of the reverse list (*|'a'b'c).

4.13 or \Rightarrow x|y

The greater of x and y. Max of a list can be determine by use of the adverb over.

```
3|2
3
1 2 3|4 5 6
4 5 6
101101b|000111b
101111b
|/12 2 3 10 / use over to determine the max of a list
12
```

4.14 asc(dsc) \Rightarrow < (>) x

Sort a list or dictionary in ascending (<) or descending (>) order. Applied to a list, these will return the indices to be used to achieve the sort.

```
<5 7 0 12
2 0 1 3

x@<x:5 7 0 12
0 5 7 12

d:`a`b`c!3 2 1;d
a|3
b|2
c|1

<d / a dictionary is sorted by value
c|1
b|2
a|3
```

4.15 less (more) \Rightarrow x < (>) y

Return true (1b) if x is less (more) than y else false (0b).

```

3<2
0b
2<3
1b
1 2 3<4 5 6
111b
((1 2 3);4;(5 6))<((101 0 5);12;(10 0)) / size needs to match
101
1
10
"a"<"b"
1b

```

4.16 group \Rightarrow =x

A dictionary of the distinct values of x (key) and their indices (values).

```

="banana"
a|1 3 5
b|0
n|2 4
=0 1 0 2 10 7 0 1 12
0|0 2 6
1|1 7
2|3
7|5
10|4
12|8

```

4.17 equal \Rightarrow x=y

Return true (1b) if x is equal to y else false (0b).

```

2=2
1b
2=3
0b
2=2.
1b
"banana"="abnaoo" / check strings of equal length by character
001100b
"banana"="apple" / unequal length strings error
^
:length

```

4.18 not \Rightarrow ~x

Boolean invert of x

```

~1b

```

```
0b
~101b
010b
~37 0 12
010b
```

4.19 match \Rightarrow x~y

Return true (1b) if x matches y else false (0b). A match happens if the two arguments evaluate to the same expression.

```
2~2
1b
2~3
0b
2~2.
0b
"banana"~"apple"
0b
`a`b~`a`b / different than = which is element-wise comparison
1b
`a`b=`a`b
11b
f:{x+y}
f~{x+y}
1b
```

4.20 enum \Rightarrow !x

Enumerate the domain of x. Given an integer, x, generate an integer list from 0 to x-1. Given a dictionary, x, return all key values.

```
!3
0 1 2

!`a`b`c!3 2 1
`a`b`c
```

In the case where x is a list of integers x1 x2 ... xk, generate a matrix of k rows where each row has size */x1 x2 ... xk and the elements of row i are !xi repeated as many times as necessary. Each row's repeats must fit below constant values of the row above.

```
!2 3 2 / Each row of length 2*3*2 = */2 3 2 = 12. First row values 0,1, second row va
0 0 0 0 0 0 1 1 1 1 1 1
0 0 1 1 2 2 0 0 1 1 2 2
0 1 0 1 0 1 0 1 0 1 0 1

!4 2 3
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1
0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```



```

!2 2 2
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 2 2
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

```

```

B:`b'+!16#2 / create a list of all 16-bit binary numbers from 0 to 65535 JOHN: THIS D

```

```

B[12123]      / pull the element 12,123
0010111101011011b

```

```

2/B[12123] / convert to base10 to check it's actually 12123
12123

```

4.21 key \Rightarrow x!y

Dictionary of x (key) and y (value).

```

3!7
,3!,7
`a`b!3 7
a|3
b|7
`a`b!((1 2);(3 4))
a|1 2
b|3 4

```

4.22 enlist \Rightarrow ,x

Create a list from x

```

,3
,3
,1 2 3
1 2 3
3=,3
,1b
3~,3
0b

```

4.23 cat \Rightarrow x,y

Concatenate x and y.

```
3,7
3 7
"hello"," ","there"
"hello there"
```

4.24 sort \Rightarrow ^x

Sort list or dictionary x into ascending order. Dictionaries are sorted using the keys.

```
^0 3 2 1
0 1 2 3
^b`a!((0 1 2);(7 6 5)) / sort dictionary by key
a|7 6 5
b|0 1 2
```

4.25 [f]cut \Rightarrow x^y

Cut list y by size, indices, or function x.

```
3^!20 / 3 rows with 6 elements per row so result is a matrix
0 1 2 3 4 5
6 7 8 9 10 11
12 13 14 15 16 17

2 3 7 ^!17 / left list indicates start number of each row.
,2
3 4 5 6
7 8 9 10 11 12 13 14 15 16

8 9 10 11 12^.1*!20
,0.8
,0.9
,1.
,1.1
1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9

{(x*x)within .5 1.5}^.1*!20 / JOHN I GET A SEG FAULT FOR THIS
0.8
0.9
1.
1.1
1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
```

Cut into domain and range, aka key a table (set an attribute to be a key)

```
t:[[]a:`x`y`z;b:1 20 1];t / an unkeyed table
a b
- --
```

```

x 1
y 20
z 1

kt:`a^t;kt / set `a as the key
a|b
-|--
x| 1
y|20
z| 1

(0#`)^kt / unkey the keyed table
a b
- --
x 1
y 20
z 1

```

4.26 count \Rightarrow #x

Count the number of elements in x.

```

#0 1 2 12
4
#((0 1 2);3;(4 5))
3
#`a`b!((1 2 3);(4 5 6)) / count the number of keys
2

```

4.27 [f]take \Rightarrow x#y

[f]take has a number of different actions depending on x. These include head (take from front of list), tail (take from back of list), common (take common elements), and filter (take where function is non-zero).

action	x
[take1], page 25,	positive int
[take1], page 25,	negative int
[take2], page 26,	list
[take3], page 26,	function

Postive (negative) x returns the first (last) x elements of y. If $\text{abs}[x] > \text{count}[y]$ then repeatedly pull from y.

```

3#0 1 2 3 4 5 / take first 3 elements of a list whose length is longer than
0 1 2

```

```
10 # 17 12 25 / take repeat: generate a list of length 10 with these three elements
17 12 25 17 12 25 17 12 25 17
```

```
-3#0 1 2 3 4 5 / take last elements of a list
3 4 5
```

If x is a list, then take returns any values common in both x and y.

```
(1 2 3 7 8 9)#(2 8 20) / common
2 8
```

```
(2 4 6 2 6 5 4) # (4 6 7 8 2 4 4 4) / always a substring of right argument
4 6 2 4 4 4
```

```
(4 6 7 8 2 4 4 4) # (2 4 6 2 6 5 4)
2 4 6 2 6 4
```

If x is a function, then select the values where the function is non-zero.

```
(0.5<)#10?1. / select values greater than 0.5 among randomly chosen numbers
0.840732 0.5330717 0.7539563 0.643315 0.6993048f
```

4.28 floor \Rightarrow _x

Return the integer floor of float x.

```
_3.7
3
```

4.29 [f]drop \Rightarrow x_y

[f]drop has a number of different actions depending on x. This include de-head (drop from front of list), de-tail (drop from back of list), uncommon (drop common elements), and filter (drop where function is zero).

action	x
[drop1], page 26,	positive int
[drop1], page 26,	negative int
[drop2], page 27,	list
[drop3], page 27,	function

Postive (negative) x drops the first (last) x elements of y.

```
3_0 1 2 3 4 5 / drop first three
3 4 5
```

```
-2_0 1 2 3 4 5      / drop last three
0 1 2 3
```

If x is a list, then drop returns any values from y not found in x.

```
(1 2 3 7 8 9)_(2 8 20) / uncommon
,20
```

If x is a function, then select values where the function is zero.

```
(0.5<)_10?1.      / select values that are less than or equal to 0.5
0.02049699 0.1269226
```

4.30 string \Rightarrow \$x

Cast x to string.

```
$`abc
"abc"
$4.7
"4.7"
```

4.31 cast \Rightarrow x\$y

Cast string y into type x.

```
`i$"23"
23
`f$"2.3"
2.3
`t$"12:34:56.789"
12:34:56.789
`d$"2020.04.20"
2020.04.20
```

4.32 unique+ \Rightarrow ?x

As unique+ is a '+' verb, because its use depends on whether x is a list (where ? means unique) or atom (where ? does a random number generation).

x is list: Return the unique values of the list x. The ? preceeding the return value explicitly shows that the list has no repeated values.

```
?`f`a`b`c`a`b`d`e`a
?f`a`b`c`d`e
?"banana"
?"ban"
```

x is atom: Generate x random values using a uniform distribution from [0,1) or -x random values of a normal distribution (mean=0, standard deviation=1).

```
?3      / 3 uniform
0.1654719 0.172977 0.4808443

?-3     / 3 normal
0.05384196 1.228946 -0.7144561
```

4.33 find+ ⇒ x?y

As find+ is a '+' verb its use depends on whether x is a list (find) or atom (range random).

x is list: Find the first element of x that matches y otherwise return the end of vector.

```
`a`b`a`c`b`a`a?`b
1
`a`b`a`c`b`a`a?`d / there is no element, so return the length of the list
7
0 1 2 3 4?10
5
(1;`a;"blue";7.4)?3
4
```

x is atom: Generate x random values from 0 to less than y (int, float, time, bit) or from y (list of characters or names). Random dates can be generated by using random ints and adding those to a start date.

```
3?5
0 0 2
3?5.
2.238258 3.038515 0.7879856
3?12:00:00
~05:55:27 09:46:18 10:49:18
3?2b
001b
3?"AB"
"BBA"
3?`a`b`c`d
`d`c`a
min 2020.01.01+1000?366
2020.01.01
max 2020.01.01+1000?366
2020.12.31
```

x is a special name, eg. 'json', 'csv', or 'nv'. This is further discussed in the Chapter 13 [I/O], page 63, section.

4.34 type ⇒ @x

Return the data type of x. Lower-case represents a single element while upper-case represents a list of type indicated.

```
@1
`i
@1.2
`f
@a
`s
@"a"
`c
```

```

@2020.04.20
`d
@12:34:56.789
`t
@(1;1.2;`a;"a";2020.04.20;12:34:56.789) / type of a list
`L
@'(1;1.2;`a;"a";2020.04.20;12:34:56.789) / type of elements of the list
`i`f`s`c`d`t

```

4.35 [f]at \Rightarrow x@y

Given a list x return the value(s) at index(indices) y.

```

(3 4 7 12)@2
7
`a`b`c@2
`c
((1 2);3;(4 5 6))@(0 1) / values at indices 0 and 1
1 2
3

```

If x is a function then determine the function with (at) parameter y.

```

{x*x}@3
9

(sin;cos;1*)@\:0.01
0.0099999833 0.99995 0.01

```

4.36 value \Rightarrow .x

Evaluate a string of valid k code or list of k code. Can also be used to covert from a string to a numeric, temporal or name type.

```

."3+2"
5

."20*1+!3"
20 40 60

.(*;16;3)
48

n:3;p:+(n?(+;-;*;%);1+n?10;1+n?10);p
% 6 3
* 2 7
- 5 5

.`p
2
14

```

```

0

."15"
15

."12.7"
12.7

."`abc"
`abc

```

4.37 [f]dot \Rightarrow x.y

Given list x return the value at list y. The action of dot depends on the shape of y.

- Index returns the value(s) at x at each index y, i.e. $x@y@0$, $x@y@1$, ..., $x@y@(n-1)$.
- Recursive index returns the value(s) at $x[y@0;y@1]$.
- Recursive index over returns $x[y[0;0];y[1]]$, $x[y[0;1];y[1]]$, ..., $x[y[0;n-1];y[1]]$.

ac-	@y	#y	example
-----	----	----	---------

tion			
sim-	'I	1	,2
ple in-			
dex			

sim-	'I	1	,1 3
ple in-			
dices			

re-	'L	1	0 2
cursive in-			
dex			

re-	'L	2	(0 2;1 3)
cursive in-			
dex over			

```

(3 4 7 12).,2
7
`a`b`c.,2
`c
x:(`x00`x01;`x10`x11`x12;`x20;`x30`x31`x32);x
x00 x01
x10 x11 x12
x20
x30 x31 x32

x . ,1
`x10`x11`x12
x . ,0 1 3
x00 x01

```



```
x10 x11 x12
x30 x31 x32
```

```
x . 3 1
`x31
x . (1 3;0 1)
x10 x11
x30 x31
```

If x is a function then apply the function to y.

```
(+). 3 2
5
```

```
(+;-;*;%).\: 3 2
5
1
6
1.5
```

```
a:{x+y};b:{x*sin y};a . 3 1
4
a . 3 1
4
b . 3 1
2.524413
```

```
(!).(`a`b`c;1 2 3)
a|1
b|2
c|3
```

5 Adverbs

Adverbs modify verbs to operate iteratively over nouns. This previous sentence likely will make sense only once you understand it so let's jump into an example. Imagine you have a list of 4 lists. Using **first** you will retrieve the first sub-list. In order to retrieve the first element of each sub-list you'll have to modify the function **first** with a modifier **each**.

```
x: (1 2; 3 4; 5 6; 7 8)
*x
1 2
*'x
1 3 5 7
```

Here now is a list of the adverbs and their names.

```
Adverb
f/ [scan], page 32, c/ [+join], page 32
f\ [scan], page 32, c\ [+split], page 33
f' [each], page 33, v' [+has], page 33
f': [eachp], page 34, v': [bin2], page 34
f/: [eachr], page 34, ([n;]f)/:
f\.: [eachl], page 35, ([n;]f)\.:
```

5.1 scan (over) \Rightarrow f\ x (f/x)

Compute $f[x;y]$ such that $f\ i = f[f\ i-1; x\ i]$. Scan and over are the same functions except that over only returns the last value.

Given a function of two inputs, output for each x according to...

- $f\ 0 \rightarrow x\ 0$
- $f\ 1 \rightarrow f[f\ 0; x\ 1]$
- ...
- $f\ i \rightarrow f[f\ i-1; x\ i]$
- ...
- $f\ n \rightarrow f[f\ n-1; x\ n]$

An example

```
(, \) ("a"; "b"; "c")
a
ab
abc
+\1 20 300
1 21 321
{y+10*x}\1 20 300
1 30 600
{y+10*x}/1 20 300
600
```

5.2 +join \Rightarrow c/x

Join a list of strings x using character c between each pair.

```
"-"/("Some";"random text";"plus";, ".")
"Some-random text-plus-."
```

The notation c/x converts a list of integers x via base c into base 10. In the example below, assuming 2 0 1 are digits in base 8, they convert to $1 + (0*8) + (2*8*8) = 129$

```
8/2 0 1
129
```

5.3 +split \Rightarrow c\x

Split the string x using character c to determine the locations.

```
" \"Here is a short sentence.\"
Here
is
a
short
sentence.
```

c\x splits a number x into base c digits.

```
8\129
2 0 1
```

5.4 each \Rightarrow f'x

Apply function f to each value in list x in a single thread. Multithread operation is done with [eachp], page 34.

```
*((1 2 3);4;(5 6);7) / first element of the list
1 2 3
```

```
*'((1 2 3);4;(5 6);7) / first element of each element
1 4 5 7
```

5.5 +has \Rightarrow v'x

Determine whether vector v has element x if x is an atom (i.e., scalar) or each element of x if x is a list. Similar to [in], page 58, but with the arguments reversed.

```
`a`b`c`a`b`a
1b
```

```
`a`b`c`a`b`d
0b
```

```
`a`b`c`a`b`a`b`x`y`z
11000b
```

```
(1 2;4 5 6;7 9) '(1 2;8 9)
```

10b

5.6 eachp \Rightarrow f':[x;y]

Apply f[x;y] using the prior value of y, eg. f[y@n;y@n-1]. The first value, n=0, returns f[y@0;x].

```
,':[`x;(`$"y",`$!5)]
y0 x
y1 y0
y2 y1
y3 y2
y4 y3
```

```
%':[100;100 101.9 105.1 102.3 106.1] / compute returns
1 1.019 1.031403 0.9733587 1.037146
```

```
100%':[100 101.9 105.1 102.3 106.1 / using infix notation
1 1.019 1.031403 0.9733587 1.037146
```

Additionally eachp (':) is a multithreaded (parallel) each. This will allow a function to run using multiple threads over a list of arguments.

```
\t {L:x?1.;sum L%sin L*cos L*sin L+L*L}'8#_2e7 / run using multiple threads
1655
```

```
\t {L:x?1.;sum L%sin L*cos L*sin L+L*L}'8#_2e7 / run in a single thread
4144
```

5.6.1 +bin \Rightarrow x':y

Given a sorted (increasing) list x, find the greatest index, i, where y>x[i].

```
n:exp 0.01*!5;n
1 1.01005 1.020201 1.030455 1.040811
n':1.025
2
```

5.7 eachr \Rightarrow f/:x, x f/:y, f/:[x;y]

name	f pa-	x	y
	ram-		
	eters		

[converge	ele-	n/a
over],	ment	

page 34,

[each 2	ar-	element/array
right],	ray	

page 35,

5.7.1 Converge Scan

Compute $f[x]$ where x is the previous output of f until the value converges or returns to x . The initial value of x is the input. Only the final value is returned. Intermediate values can be retrieved via [converge scan], page 35.

```
{x*x}/:0.99 / converge to value
0.
```

```
x:3 4 2 1 4 99
x/:0
4
```

```
/ the latter example worked out manually. Recall that x 3 is the same as x[3]
x 0
3
x 3
1
x 1
4
x 4 / x[4]=4 so will stop
4
```

5.7.2 Each Right

Apply $f[x;y]$ to each value of y , i.e. the right one.

```
(!2)+/:20+!4
20 21
21 22
22 23
23 24
```

5.8 eachl \Rightarrow $f\backslash:x$, $x\ f\backslash:y$, $f\backslash:[x;y]$

name fpa- x y
ram-
eters

[converge ele- n/a
 scan], ment
 page 35,

[each 2 ar- element/array
 left], ray
 page 36,

5.8.1 Converge Scan

Compute $f[x]$ where x is the previous output of f until the value converges or returns to x . The initial value of x is the input

```
{x*x}\:0.99 / converge to value
0.99 0.9801 0.9605961 ... 1.323698e-18 1.752177e-36 0.
```

```

x:3 4 2 1 4 99 / until output is x
x\:0
0 3 1 4

/ the latter example worked out manually
x 0
3
x 3
1
x 1
4
x 4          / x[4]=4 so will stop
4

```

5.8.2 Each Left

Apply `f[x;y]` to each value of `x`, i.e. the left one.

```

(!2)+\:30+!4
30 31 32 33
31 32 33 34

+\: [!2;!4]
0 1 2 3
1 2 3 4

```

6 Noun

The basic data types of the k9 language are booleans, numbers (integer and float), text (characters and enumerated/name) and temporal (date and time). It is common to have functions operate on multiple data types.

In addition to the basic data types, data can be put into lists (uniform and non-uniform), dictionaries (key-value pairs), and tables (transposed/flipped dictionaries). Dictionaries and tables will be covered in a separate chapter.

The set of k9 data, aka nouns, are as follows.

```
type
[bool], page 38, 011b
[int], page 38, 0N 0 2 3
[flt], page 39, 0n 0 2 3.4
[char], page 39, " ab"
[name], page 39, ``ab
[uuid], page 40

[date], page 40, 2024.01.01
[time], page 40, 12:34:56.123456789
```

Data types can be determined by using the @ function on values or lists of values. In the case of non-uniform lists @ returns the type of the list `L but the function can be modified to evaluate each type @' instead and return the type of each element in the list.

```
@1          / integer atom
`i
@1 2 3      / integer list
`I
@12:34:56.789 / time atom
@(3;3.1;"b";`a;12:01:02.123;2020.04.05) / mixed list
`L
@'(3;3.1;"b";`a;12:01:02.123;2020.04.05)
`i`f`c`n`t`d
```

6.1 Atom Types

This section lists all the different types available. Generally lower case specifies atoms and upper case as lists.

name	type	example	note
b	boolean	1b	
c	character	"a"	
d	date	2020.06.14	
e	float	3.1	
f	float	3.1f	
g	int	2g	1 byte unsigned
h	int	2h	2 byte unsigned
i	int	2	4 byte unsigned

j	int	2j	8 byte signed
n	name	'abc	8 char max
s	time	12:34:56	second
S	datetime	2020.06.15T12:34:56	second
t	time	12:34:56.123	millisecond
T	datetime	2020.06.15T12:34:56.123	millisecond
u	time	12:34:56.123456	microsecond
U	datetime	2020.06.15T12:34:56.123456	microsecond
v	time	12:34:56.123456789	nanosecond
V	datetime	2020.06.15T12:34:56.123456789	nanosecond

6.2 bool \Rightarrow Boolean b

Booleans have two possible values 0 and 1 and have a 'b' to avoid confusion with integers, eg. 0b or 1b.

```
0b
0b
1b
1b
10101010b
10101010b
```

6.2.1 boolean logic

k9 implements logic operations in the usual way.

```
x:0101b;y:0011b
```

```
[[x:x;y:y]AND:x&y;OR:x|y;NAND:~x&y;NOR:~x|y;XOR:~x=y;XNOR:x=y]
x y|AND OR NAND NOR XOR XNOR
- -|--- -- ---- - - - -
0 0| 0 0 1 1 0 1
1 0| 0 1 1 0 1 0
0 1| 0 1 1 0 1 0
1 1| 1 1 0 0 0 1
```

6.3 Numeric Data

Numbers can be stored as integers and floats.

6.3.1 int \Rightarrow Integer g, h, i, j

Integers can be stored in four different ways which correspond 1, 2, 4, and 8 bytes. The first three are unsigned and the last (j) is signed. Positive numbers default to i and negative and very large numbers default to j. One can specify a non-default type by adding one of the four letters immediately following the number.

```
@37    / will default to i
`i
@-37   / negative so will default to j
`j
```



```

    @37g    / cast as g
    `g

    b: {-1+*/x#256}
    `g b[1]
    255g
    `h b[2]
    65535h
    `i b[4]
    4294967295
    `j b[7]
    72057594037927935

```

6.3.2 flt \Rightarrow Float e, f

Float

```

    3.1
    3.1
    3.1+1.2
    4.3
    3.1-1.1
    2.
    @3.1-1.1
    `e
    @3.1
    `e
    a:3.1;
    @a
    `e
    @1%3
    `f

```

6.4 Text Data

Text data come in characters, lists of characters (aka strings) and enumerated types. Enumerated types are displayed as text but stored internally as integers.

6.4.1 char \Rightarrow Character c

Characters are stored as their ANSI value and can be seen by conversion to integers. Character lists are equivalent to strings.

```

    @"b"
    `c
    @"bd"
    `C

```

6.4.2 name \Rightarrow Name n

Names are enumerate type shown as a text string but stored internally as a integer value.

```
@`blue
`n
@`blue`red
`N
```

6.5 Unique Identifier

TBD

6.5.1 uuid \Rightarrow Uuid

TBD

6.6 Temporal Data

Temporal data can be expressed in time, date, or a combined date and time.

6.6.1 time \Rightarrow Time s, t, u, v

Time has four types depending on the level of precision. The types are seconds (s), milliseconds (t), microseconds (u), and nanoseconds (v). The times are all stored internally as integers. The integers are the number of time units. For example 00:00:00.012 and 00:00:00.000000012 are both stored as 12 internally.

```
@12:34:56.789          / time
`t
.z.t                  / current time in GMT
17:32:57.995
(t:.z.t)-17:30:00.000
00:03:59.986
t
17:33:59.986
`i 00:00:00.001        / numeric representation of 1ms
1
`i 00:00:01.000        / numeric representation of 1s
1000
`i 00:01:00.000        / numeric representation of 1m
60000
`t 12345               / convert milliseconds to time
00:00:12.345
```

6.6.2 date \Rightarrow Date d

Dates are in yyyy.mm.dd format. Dates are stored internally as integers with 0 corresponding to 2001.01.01.

```
@2020.04.20           / date
`d
.z.d                  / current date in GMT
2020.12.05
`i .z.d               / numeric representation of date
7278
```

```

`i 2001.01.01          / zero date
0
`d 0                   / zero date
2001.01.01

```

6.6.3 datetime ⇒ Datetime d

Dates and times can be combined into a single datetime element by combining a date, the letter T, and the time together without spaces. The datetime use the same lettering as the time precision but in uppercase. Datetimes are stored internally as integers. For example 2001.01.02T00:00:00.000 is stored as 86,400,000, the number of milliseconds in a day.

```

@2020.04.20T12:34:56.789 / date and time
`d
`T$"2020.04.20 12:34:56.789" / converting from string
2020.04.20T12:34:56.789

```

6.7 Extreme values

Data types can represent in-range, null, and out-of-range values.

type	null	out of range
i	0N	0W
f	0n	0w
	0%0	
	0n	
	1e500	
	0w	

7 List

Lists and derivatives of lists are core to k9 which likely is not a surprise given that the language is made to process large quantities of data. Performance will be best when working with uniform lists of a single data type but k9 supports list of non-uniform type also.

Lists are automatically formed when a sequence of uniform type are entered or generated by any function.

```
1 3 12      / list of ints
1 3 12

3.1 -4.1 5. / list of floats
3.1 -4.1 5.

"abc"      / list of chars
"abc"

`x`y`z     / list of names
`x`y`z
```

In order to determine if data is an atom or list one can use the [type], page 28, command. The command returns a lower case value for atoms and an upper case value for lists.

```
@1          / an integer
i

@1 3 12     / list of ints
I

@,1         / list of single int via [enlist], page 23
I
```

Commands that generate sequences of numbers return lists regardless if the count is 1 or many.

```
@!0
`I
@!1
`I
@!2
`I
```

7.1 List Syntax

In general, lists are created by data separated by semicolons and encased by parenthesis.

```
(1;3;12)      / list of ints
@ (1;3.;`a;"b") / non-uniform list
L
@((1;3);(12;0)) / list of lists
L
```

```

@'((1;3);(12;0)) / each list is type I
`I`I
,,,,,(3;1)      / a list of a list of a list ...
,,,,,3 1

```

7.2 List Indexing

Lists can be indexed by using a few notations. The @ notation is often used as it's less characters than [] and the explicit @ instead of space is likely more clear.

```

a:2*1+!10 / 2 4 ... 20
a[9]      / square bracket
20
a@9       / at
20
a 9       / space
20
a(9)      / parenthesis
20
a[10]     / out of range return zero
0

```

7.3 Updating List Elements

Lists can be updated element wise by specifying setting the indexed element to a required value. There is also a syntax for updating many elements and that it found at [amend], page 60.

```

a:2*1+!10;a
2 4 6 8 10 12 14 16 18 20
a[3]:80
a
2 4 6 80 10 12 14 16 18 20

```

7.4 Function of Two Lists

This section will focus on functions (f) that operate on two lists (x and y). As these are internal functions examples will be shown with infix notation (x+y) but prefix notation (+[x;y]) is also permissible.

7.4.1 Pairwise

These functions operate on list elements pairwise and thus requires that x and y are equal length.

- x+y : Add
- x-y : Subtract
- x*y : Multiply
- x%/y : Divide
- x&y : AND/Min

- `x|y` : OR/Max
- `x>y` : Greater Than
- `x<y` : Less Than
- `x=y` : Equals
- `x!y` : Dictionary
- `x$y` : Sumproduct

```

x:1+!5;y:10-2*!5
x
1 2 3 4 5
y
10 8 6 4 2
x+y
11 10 9 8 7
x-y
-9 -6 -3 0 3
x*y
10 16 18 16 10
x%y
0.1 0.25 0.5 1 2.5f
x&y
1 2 3 4 2
x|y
10 8 6 4 5
x>y
00001b
x<y
11100b
x=y
00010b
x!y
1|10
2| 8
3| 6
4| 4
5| 2
x$y
70

```

7.4.2 Each Element of One List Compared to Entire Other List

These functions compare `x[i]` to `y` or `x` to `y[i]` and `f` is not symmetric to its inputs, i.e. `f[x;y]` does not equal `f[y;x]`;

- `x^y` : Reshape all element in `y` by `x`
 - `x#y` : List all elements in `x` that appear in `y`
 - `x?y` : Find all elements in `y` from `x`
- ```

x:0 2 5 10

```

```

y:!20
x^y
0 1
2 3 4
5 6 7 8 9
10 11 12 13 14 15 16 17 18 19

```

```

x:2 8 20
y:1 2 3 7 8 9
x#y
2 8
x?y
3 0 3 3 1 3

```

### 7.4.3 Each List Used Symmetrically

These functions are symmetric in the inputs  $f[x;y]=f[y;x]$  and the lists are not required to be equal length.

- `x_y` : Values present in only one of the two lists

```

x:2 8 20
y:1 2 3 7 8 9
x_y
1 3 7 9

```

## 8 Dictionary

Dictionaries are a data type of key-value pairs typically used to retrieve the value by using the key. In other computer languages they are also known as associative arrays and maps. Keys should be unique to avoid lookup value confusion but uniqueness is not enforced. The values in the dictionary can be single elements, lists, tables, or even dictionaries.

Dictionaries in k9 are often used. As an example in the benchmark chapter the market quote and trade data are dictionaries of symbols (name keys) and market data (table values).

### 8.1 Dictionary Creation $\Rightarrow$ `x!y` or `[x;y]`

Dictionaries are created by using the key symbol or square bracket notation and listing the keys (x) and values (y).

```
x!y
 d0:`a37!12;d0 / single entry
a37|12

 d1:`pi`e`c!3.14 2.72 3e8;d1 / elements
pi|3.14
e |2.72
c |3e+08

 `a`b`c!(1 2 3;10 20 30;100 200 300)
a|1 2 3
b|10 20 30
c|100 200 300

[x:y]
 d0:[a37:12]
a37|12

 d1:[pi:3.14;e:2.72;c:3e8];d1
pi|3.14
e |2.72
c |3e+08

 [a:1 2 3;b:10 20 30;c:100 200 300]
a|1 2 3
b|10 20 30
c|100 200 300
```

### 8.2 Dictionary Indexing $\Rightarrow$ `x@y`

Dictionaries, like lists, can be indexed in a number of ways.

```
x:`a`b`c!(1 2;3 4;5 6);x
a|1 2
b|3 4
```



```

c|5 6
 x@`a
1 2
 x@`a`c
1 2
5 6
/ all these notations for indexing work, output suppressed
x@`b; / at
x(`b); / parenthesis
x `b; / space
x[`b]; / square bracket

```

### 8.3 Dictionary Key $\Rightarrow$ !x

The keys from a dictionary are retrieved by using the ! function.

```

!d0
`pi`e`c
!d1
`time`temp
!d2
0 10 1

```

### 8.4 Dictionary Value $\Rightarrow$ x[]

The values from a dictionary are retrieved by bracket notation.

```

d0[]
3.14 2.72 3e+08
d1[]
12:00 12:01 12:10
25. 25.1 25.6

d2[]
37.4 46.3 0.1

```

One could return a specific value by indexing into a specific location. As an example in order to query the first value of the temp from d1, one would convert d1 into values (as value .), take the second index (take the value 1), take the second element (take the temp 1), and then query the first value (element 0).

```

d1
time|12:00 12:01 12:10
temp|25 25.1 25.6

d1[]
12:00 12:01 12:10
25 25.1 25.6

d1[] [1]
25 25.1 25.6

```

```
d1[][1;0]
25.
```

## 8.5 Sorting a Dictionary by Key $\Rightarrow$ ^x

```
d0
pi|3.14
e |2.72
c |3e+08
```

```
^d0
c |3e+08
e |2.72
pi|3.14
```

## 8.6 Sorting a Dictionary by Value $\Rightarrow$ <x (>x)

```
d0
pi|3.14
e |2.72
c |3e+08
```

```
<d0
e |2.72
pi|3.14
c |3e+08
```

```
>d0
c |3e+08
pi|3.14
e |2.72
```

## 8.7 Flipping a Dictionary into a Table $\Rightarrow$ +x

This command flips a dictionary into a table but will be covered in detail in the table section.

```
d1
time|12:00 12:01 12:10
temp|25 25.1 25.6
```

```
+d1
time temp

12:00 25
12:01 25.1
12:10 25.6
```

```
d1~+d1
```

0b

## 8.8 Functions that operate on each value in a dictionary

There a number of simple functions on dictionaries that operate on the values. If 'f' is a function then f applied to a dictionary return a dictionary with the same keys and the values are application of 'f'.

- -d : Negate
- d + N : Add N to d
- d - N : Subtract N from d
- d \* N : Multiple d by N
- d % N : Divide d by N
- |d : Reverse
- <d : Sort Ascending
- >d : Sort Descending
- ~d : Not d
- &d : Given d:x!y output each x, y times, where y must be an integer
- =d : Given d:x!y y!x

Examples

```
d2
0|37.4
10|46.3
1|0.1
```

```
-d2
0|-37.4
10|-46.3
1|-0.1
```

```
d2+3
0|40.4
10|49.3
1|3.1
```

```
d2-1.7
0|35.7
10|44.6
1|-1.6
```

```
d2*10
0|374
10|463
1|1
```

```
d2%100
0|0.374
10|0.463
1|0.001
```

## 8.9 Functions that operate over values in a dictionary

There are functions on dictionaries that operate over the values. Given function `f` applied to a dictionary `d` then `f d` returns a value.

- `*d` : First value

```
d0
pi|3.14
e |2.72
c |3e+08
```

```
*d0
3.14
```

## 9 User Functions

User-defined functions are used to define a set of operations to be easily called. As these functions are built on top of the fast internal functions of k9 they should be performant also.

Multi-line functions must be defined with an initial space on all lines except the first. These can be loaded into memory by using the [load], page 80, command and then used by calling by name.

```
Func {[a;b]a+b}
```

### 9.1 Function arguments

Functions default to arguments of x, y, and z for the first three parameters but this can be explicitly added or modified as needed. Given k9's terseness it's rare to see good k9 code with long variable names.

```
f1:{x+2*y} / implicit arguments
f1[2;10]
22

f2:{[x;y]x+2*y} / explicit
f2[2;10]
22

f3:{[g;h]g+2*h} / explicit and renamed
f3[2;10]
22

f4:{[please;dont]please+2*dont} / ouch
f4[2;10]
22

@f1
\.
```

### 9.2 Function Defitions

Functions can have intermediate calculations and local variables. Function local variable do not collide with global variables.

```
a:3;a
3

f:{a:12;x:sqrt x;x+a}

f 10
15.16228

a
```

3

### 9.3 Function Return

Function return the last value in the definition unless the definition ends with a semicolon in which case the function returns nothing.

```
f:{x+2;};f 10
```

```
f:{x+2;27};f 10
27
```

```
f:{x+2};f 10
12
```

### 9.4 Calling functions

Functions, like lists, can be called with a number of notations. Typically one uses square bracket notation when the function takes multiple arguments. If a function is called with less than the required number of arguments then it will return a function that requires the remaining arguments.

```
f1:{[x] x}
f2:{[x;y] (x;y)}
f3:{[x;y;z] (x;y;z)}
```

```
f1[`a]
`a
```

```
f2[37;`a]
37
a
```

```
f3["hi";37;`a]
hi
37
a
```

```
f2[37]
{[x;y] (x;y)}[37]
f2[;`a]
{[x;y] (x;y)}[;`a]
```

```
f1[`a]
`a
f1 `a
`a
f1@`a
`a
```

## 9.5 Anonymous functions

Functions not set with a function name are anonymous.

```
{x+2*y}[2;10]
22
```

## 9.6 Recursive functions

k9 allows one to define a function which calls itself. Care should be taken to avoid an infinite loop.

```
f:{$[x>1;x*f@x-1;1]};f 5
120
```

## 9.7 Chained functions

It may be necessary to define a function to call a function without specifying arguments. Imagine this trivial case.

```
f1:{!x}
f2:{x+2} f1
{x+2}
^
:rank
```

In order to succeed f1 needs to have an @ in the definition of f2. This is only required when the function calls a function that is not completely specified with input parameters.

```
f1:{!x}
f2:{x+2} f1@
f2 3
2 3 4
```

## 10 Expression Evaluation

k9 has a compact way to evaluate an expression on data structures including dictionaries and tables.

Expr :a+b

### 10.1 `expr ⇒ x :y`

Evaluate expression(s) `y` on table or dictionary `x`. The space between `x` and `:` is required otherwise it becomes `[set]`, page 16.

```
d:`city!`London
d :city
`London
```

```
d :`Paris=city
0b
```

```
d:`city!`London`Paris`Madrid
d :`Paris=city
010b
```

```
d:`a`b`c!(3 2 1;1 2 3;1 2 3);d
a|3 2 1
b|1 2 3
c|1 2 3
```

```
d :a+b*c
4 6 10
```

```
t:+d;t
a b c
- - -
3 1 1
2 2 2
1 3 3
```

```
t : (a+b*c)
4 6 10
```

```
d:`a`b`c!(3 2 1;1 2 3;1 2 3);d
d : (100*a;b+c)
300 200 100
2 4 6
```



## 11 Named Functions

This chapter covers the non-symbol named functions. This includes some math (eg. sqrt but not +), wrapper (eg. count for #) and range (eg. within) functions.

[exp], page 55, [log], page 55, [sin], page 55, [cos], page 55, [sqr], page 55, [sqrt], page 55, [div], page 55, [mod], page 56, [bar], page 56; [topX], page 56, [freq], page 56, [msum], page 57, [mavg], page 57, [sums], page 57, [deltas], page 57, [prm], page 57, [cmb], page 57

### 11.1 Math Functions $\Rightarrow$ exp log sin cos sqr sqrt div mod bar

#### 11.1.1 exp $\Rightarrow$ exp x

```
exp 1
2.718282
```

#### 11.1.2 log $\Rightarrow$ log x

Log computes the natural log.

```
log 10
2.302585
```

#### 11.1.3 sin $\Rightarrow$ sin x

sin computes the sine of x where x is in radians.

```
sin 0
0f
sin 3.1416%2
1.
```

#### 11.1.4 cos $\Rightarrow$ cos x

cos computes the cosine of x where x is in radians.

```
cos 0
1f
cos 3.1416%4
0.7071055
```

#### 11.1.5 sqr $\Rightarrow$ sqr x

```
sqr 2
4.0
```

#### 11.1.6 sqrt $\Rightarrow$ sqrt x

```
sqrt 2
1.414214
```

#### 11.1.7 div $\Rightarrow$ x div y

y divided by x using integer division. x and y must be integers.

```
2 div 7
3
```

```
5 div 22
4
```

### 11.1.8 mod $\Rightarrow$ x mod y

The remainder after y divided by x using integer division. x and y must be integers.

```
12 mod 27
3
5 mod 22
2
```

### 11.1.9 bar $\Rightarrow$ x bar y

For each value in y determine the number of integer multiples of x that is less than or equal to each x.

```
10 bar 9 10 11 19 20 21
0 10 10 10 20 20
```

## 11.2 Helper Functions $\Rightarrow$ top freq msum mavg sums deltas prm cmb

### 11.2.1 top $\Rightarrow$ top x

Return the index of the max value of list x.

```
top 3 13 15 1 17 0 -3
4

top "abzccdefeajfekjrlke"
2
```

### 11.2.2 freq $\Rightarrow$ freq x

Compute frequency of the items in list x.

```
freq "abzccdefeajfekjrlke"
a|2
b|1
c|2
d|1
e|4
f|2
j|2
k|2
l|1
r|1
z|1

freq 1000?10
0| 95
1|106
```

```

2| 84
3| 98
4|109
5|104
6| 94
7| 95
8|120
9| 95

```

### 11.2.3 msum $\Rightarrow$ x msum y

Compute the x wide moving sum of list y.

```

3 msum !10
0 1 3 6 9 12 15 18 21 24

```

### 11.2.4 mavg $\Rightarrow$ x mavg y

Compute the x wide moving average of list y.

```

3 mavg !10
0 0.3333333 1 2 3 4 5 6 7 8

```

### 11.2.5 sums $\Rightarrow$ sums x

Compute the sum of list x. Sames as +\.

```

sums !10
0 1 3 6 10 15 21 28 36 45

```

### 11.2.6 deltas $\Rightarrow$ deltas x and x deltas y

Compute the difference between each element in list x and the previous value. If delta is called with two parameters then x will be used as the first value to delta instead of the default 0.

```

deltas 1 5 10
1 4 5

```

```

1 deltas 1 5 10
0 4 5

```

### 11.2.7 prm $\Rightarrow$ prm x

Determine all permutations of up x-1.

```

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

```

### 11.2.8 cmb $\Rightarrow$ x cmb y

Determine all x value combinations up to y.

```

2 cmb 4
2 3

```

```

1 3
1 2
0 3
0 2
0 1

`a`b`c`d@2 cmb 4 / choose from `a`b`c`d
`c`d
`b`d
`b`c
`a`d
`a`c
`a`b

```

### 11.3 Range Functions $\Rightarrow$ key unkey meta in bin within

#### 11.3.1 key $\Rightarrow$ x key y

Key table y with x.

```

`a key [[a:`x`y;b:1 2]
a	b
x|1
y|2

```

#### 11.3.2 unkey $\Rightarrow$ unkey x

Remove key from table x.

```

unkey [[a:`x`y;b:1 2]
a b
- -
x 1
y 2

```

#### 11.3.3 meta $\Rightarrow$ meta

Determine the types of the fields in a table.

```

meta [[a:`x`y;b:1 2]
a|n
b|i

```

#### 11.3.4 in $\Rightarrow$ x in y

Determine if x is in list y. Similar to [+has], page 33, but arguments reversed.

```

`b in `a`b`d`e
1b
`c in `a`b`d`e
0b

```

**11.3.5 bin  $\Rightarrow$  x bin y**

Given a sorted (increasing) list x, find the greatest index, i, where  $y > x[i]$ .

```
n:exp 0.01*!5;n
1 1.01005 1.020201 1.030455 1.040811
1.025 bin n
2
```

**11.3.6 within  $\Rightarrow$  x within y**

Test if x is equal to or greater than  $y[0]$  and less than  $y[1]$ .

```
3 within 0 12
1b
```

```
0 within 0 12
1b
```

```
12 within 0 12
0b
```

```
23 within 0 12
0b
```

## 12 Knit Functions

These functions modify lists and dictionaries given a list of indices and functions or values to replace.

`@[r;i;f[;y]]` [amend], page 60

`.[r;i;f[;y]]` [dmend], page 61

### 12.1 amend $\Rightarrow$ @[r;i;f[;y]]

Replace the values in list/dictionary `r` at indices `i` with element `f` or function `f` and parameter `y`. The original list is not modified.

| action             | f        | y            |
|--------------------|----------|--------------|
| [amend1], page 60, | element  | n/a          |
| [amend2], page 60, | :        | element/list |
| [amend3], page 60, | function | n/a          |
| [amend4], page 61, | function | 2nd param    |

Amend to element.

```
r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
@[r;0 3;29]
29
2 3
4 5
29
```

Amend with element/array. If using an array, then the `i` and `y` must be arrays of equal length.

```
r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
@[r;1 2;;;(0;3 5)]
0 1
0
3 5
6
```

Amend with function `f[r]` at indices `i`.

```
r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
@[r;1 2;sqrt]
0 1
1.414214 1.732051
2.0 2.236068
6
```

Amend with function `f[r;y]` at indices `i`.

```
r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
@[r;1 2;*;10 100]
0 1
20 30
400 500
6
```

## 12.2 `dmend` $\Rightarrow$ `.[r;i;f[y]]`

Similar to `[amend]`, page 60, but using `i` to fully index `r`.

| action                           | f        | y         |
|----------------------------------|----------|-----------|
| <code>[dmend1]</code> , page 61, | element  | n/a       |
| <code>[dmend3]</code> , page 62, | function | n/a       |
| <code>[dmend4]</code> , page 62, | function | 2nd param |

Dmend to element.

```
r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
.[r;0 1;12]
0 12
2 3
```

```
4 5
6
```

Dmend with function  $f[r]$  at indices  $i$ .

```
 r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
 .[r;1 1;sqrt]
0 1
2.0 1.732051
4 5
6
```

Dmend with function  $f[r;y]$  at indices  $i$ .

```
 r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
 .[r;1 1;+;100]
0 1
2 103
4 5
6
```



## 13 I/O and Interface

Given k9 is useful for analyzing data it won't be a surprise that input and output (I/O) are supported. k9 has been optimized to read in data quickly so if you have a workflow of making a tea while the huge csv file loads you might have an issue.

k9 is also able to interface with other languages so one doesn't need to rewrite everything already written into k9.

### I/O

```
0: [r line], page 65/[w line], page 65, line
1: [r byte], page 65/[w byte], page 65, char
*2: [r data], page 65/[w data], page 66, data
*3: kipc set
*4: http get
```

### Interface

```
`csv?`csv t / [read csv], page 64/[write csv], page 64, csv
`json?`json t / [read json], page 64/[write json], page 64, json
*ffi: "./a.so"5:`f!"ii" /I f(I i){return 2+i;} //cblas ..
*c/k: "./b.so"5:`f!1 /K f(K x){return ki(2+xi);} //feeds ..
```

### 13.1 Example of Data I/O

Let's begin with a small trivial example to show how to read data from a csv file. We'll generate a small table and save it with and without headers (wHeader.csv and woHeader.csv respectively). Then we'll read it back in specifying types and with the csv reader. Types are specified by one letter and the full list can be found in [Atom Types], page 37.

```
t:[[]a:3 2 1;b:1. 2. 4.;c:`x`y`z];t / generate a table of data
a b c
- - -
3 1. x
2 2. y
1 4. z
```

```
"wHeader.csv"0:`csv t / save with column headers
wHeader.csv
```

```
"woHeader.csv"0:1_`csv t / save without col headers
woHeader.csv
```

```
`csv?0:"wHeader.csv" / default reader
a b c
- - -
3 1. x
2 2. y
1 4. z
```

```

 ("",";"ifn")0:"wHeader.csv" / specify separator and types
a b c
- - - -
3 1. x
2 2. y
1 4. z

 (`e`f`g;"",";"ifn")0:"woHeader.csv" / speicfy names, separator and types
e f g
- - - -
3 1. x
2 2. y
1 4. z

```

### 13.2 read csv $\Rightarrow$ 'csv?x

Convert x from csv format. Works on lists and tables.

```

`csv?("a,b";"1,3."; "2,4.")
a b
- --
1 3.
2 4.

```

### 13.3 write csv $\Rightarrow$ 'csv x

Convert x to CSV format. Works on lists and tables.

```

`csv [[]a:1 2;b:3. 4.]
a,b
1,3.
2,4.

```

### 13.4 read json $\Rightarrow$ 'json?x

Convert x from json format. Works on lists and tables.

```

`json?("{\"a:1,b:3.\"";"{a:2,b:4.\"")
a b
- --
1 3.
2 4.

```

### 13.5 write json $\Rightarrow$ 'json x

Convert x to json format. Works on lists and tables.

```

`json [[]a:1 2;b:3. 4.]
{a:1,b:3.}
{a:2,b:4.}

```

**13.6 read name value  $\Rightarrow$  'nv?x**

Convert x, a list of name:value pairs, to a dictionary.

```
`nv?("key1:value1";"key2:value2")
key1|value1
key2|value2
```

**13.7 write line  $\Rightarrow$  x 0:y**

Output to x the list of strings in y. y must be a list of strings. If y is a single stream then convert to list via enlist.

```
""0:("blue";"red") / "" represents stdout
blue
red

"some.csv" 0:`csv [[]a:1 2;b:3. 4.]
"some.csv"
```

**13.8 read line  $\Rightarrow$  0:x**

Read from file x.

```
0:"some.csv"
a,b
1,3.
2,4.
```

**13.9 write byte  $\Rightarrow$  x 1:y**

Output to file x the list of chars in y as bytes.

```
"some.txt"1:"0123ABab"
"some.txt"

t:1:"some.txt";t
48 49 50 51 65 66 97 98g

`c t
"0123ABab"
```

**13.10 read byte  $\Rightarrow$  1:x**

Read from byte data from file x. See [w byte], page 65, for an example to write and then read byte data.

**13.11 read data  $\Rightarrow$  2: x**

Enterprise Only

Load file, eg. csv or from a (x 2: y) save. For the latter one can find a “save then load” example in the next section.

```
2:`t.csv
```

```

s t e p z

AABL 09:30:00 D 11 4379
AABL 09:30:00 B 40 3950

2:`r / read from file
a b c d e

0.5366064 0.8250996 0.8978589 0.4895149 0.6811532
0.1653467 0.05017282 0.4831432 0.4657975 0.4434603
0.08842649 0.8885677 0.23108 0.3336785 0.6270692
..

```

### 13.12 write data $\Rightarrow$ x 2: y

Enterprise only

Save to file x non-atomic data y (eg. lists, dictionaries, or tables).

This example saves 100mio 8-byte doubles to file. The session is then closed and a fresh session reads in the file. Both the write (420 ms) and compute statistic from file (146 ms) have impressive speeds given the file size (800 MB).

```

n:_1e8
r:+`a`b`c`d`e!5^n?1.;r
`r 2:r / write to file

```

Start new session.

```

\t r:2:`r;select avg a,sum b, max c, min d, sum e from r
148

```

### 13.13 conn/set $\Rightarrow$ 3:

Enterprise only

### 13.14 http/get $\Rightarrow$ 4:

Enterprise only

### 13.15 lib $\Rightarrow$ 5:

Enterprise only

Load shared library.

Contents of file 'a.c'

```

int add1(int x){return 1+x;}
int add2(int x){return 2+x;}
int indx(int x[],int y){return x[y];}

```

Compile into a shared library (done on macos here)

```

% clang -dynamiclib -o a.so a.c

```

Load the shared library into the session.

```
f:"./dev/a.so"5:{add1:"i";add2:"i";indx:"Ii"}
f[`add1] 12
13
f[`indx][12 13 14;2]
14
```

## 14 Tables

k9 has the ability to store data in a tabular format containing named columns and rows of information as tables. If the data to be stored and queried is large then tables should be the default data type to be considered. This chapter introduces the different types of data tables available in k9. Table, utable and xtable are very similar and as you'll see in the Chapter 15 [kSQL], page 71, chapter are easy to query. In the Chapter 3 [Benchmark], page 12, chapter, tables were shown to be fast to save, read, and query.

```
[t], page 68: [[i:2 3;f:3 4.;s:`a`b]
[utable], page 69, [[b:..]a:..]
[xtable], page 69, `..![[a:..]
```

### 14.1 table

The table is the most basic of the three types. A table consists of columns and rows of information where each column has a name. Tables can be created in three different ways (1) specification via table format, (2) flipping a dictionary, or (3) reading in from a file.

#### 14.1.1 Table format

Tables can be created with the table square bracket notation.

As an example let's create a table with two columns named "a" and "col2" with three rows. The syntax is to surround the definition with square brackets and then have a first element of empty square brackets. In general this in square bracket pair will contain any keys but more on this will happen in utable. After that it's first column name, colon, and the list of values, then second column, and continuing for all the columns.

```
[[a:1 20 3; col2: 3.6 4.8 0.1]
a col2
-- ----
1 3.6
20 4.8
3 0.1

[[a:1; col2:3.6] / will error :class as lists required
[[a:1; col2:3.6]

:class

[[a:,1; col2:,3.6] / using enlist will succeed
[[a:,1;col2:,3.6]
```

#### 14.1.2 Dictionary format

Tables can also be created by flipping a dictionary into a table.

```
+`a`col2!(1 20 3; col2: 3.6 4.8 0.1)
a col2
-- ----
1 3.6
```

```
20 4.8
3 0.1
```

### 14.1.3 File import

Tables can also be created by reading in a file.

```
t.csv
a, col2
1, 3.6
20, 4.8
3, 0.1
```

Use load file 2:x which returns a table.

```
2:`t.csv
a col2
-- ----
1 3.6
20 4.8
3 0.1
```

## 14.2 xtable

A cross tab (xtable) is a collection of tables stored in a dictionary where the keys are symbols and the values are tables. Below is an example where the keys are symbols and the values are end-of-day prices.

```
x1:+`d`p!(2020.09.08 2020.09.09 2020.09.10;140 139 150)
x2:+`d`p!(2020.09.08 2020.09.10;202 208)
eod:`AB`ZY!(x1;x2)
eod`AB
d p

2020.09.08 140
2020.09.09 139
2020.09.10 150
```

## 14.3 utable

utable (or key table) is a table where some of the columns are keyed and thus should not have duplicate values.

```
[[d:2020.09.08 2020.09.09 2020.09.10]p:140 139 150]
d	p
2020.09.08|140
2020.09.09|139
2020.09.10|150

`d` [[d:2020.09.08 2020.09.09 2020.09.10;p:140 139 150]
d |p
```

-----|---  
2020.09.08|140  
2020.09.09|139  
2020.09.10|150



## 15 kSQL

kSQL is a powerful query language for tables. It has similarities to ANSI SQL but additional features to make it easier to work with data.

```
Database
[select], page 71, [A], page 71, [by B], page 73, from T [where C], page 73; [update],
page 73, A from T
x,y / [insert], page 78, [upsert], page 79, [union], page 75, equi-and-asof [left,
page 75
x+y / equi-and-asof outerjoin (e.g. combine markets through time)
x#y / take/intersect innerjoin (x_y for drop/difference)
```

### 15.1 Queries

#### 15.1.1 select

There are a number of ways to return a complete table with kSQL. You can use the table name, a kSQL query without columns, or a fully specified query with columns.

```
n:5;t:[[]x:!n;y:sin !n]
t
x y
- -----
0 0.
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025
```

```
select from t
x y
- -----
0 0.
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025
```

```
select x,y from t
x y
- -----
0 0.
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025
```

### 15.1.2 A

A is the list of fields to return from the select query. If A is missing then all fields will be returned. A can create new column names. A is probably called A because it's before B and C.

```
n:5;t:[[]x:!n;y:sin !n];t
```

```
x y
```

```
- -----
```

```
0 0.
```

```
1 0.841471
```

```
2 0.9092974
```

```
3 0.14112
```

```
4 -0.7568025
```

```
select from t
```

```
x y
```

```
- -----
```

```
0 0.
```

```
1 0.841471
```

```
2 0.9092974
```

```
3 0.14112
```

```
4 -0.7568025
```

```
select x from t
```

```
x
```

```
-
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
select y,x from t / reorder columns
```

```
y x
```

```
----- -
```

```
0. 0
```

```
0.841471 1
```

```
0.9092974 2
```

```
0.14112 3
```

```
-0.7568025 4
```

```
select x,z:y from t / create new column z
```

```
x z
```

```
- -----
```

```
0 0.
```

```
1 0.841471
```

```
2 0.9092974
```

```

3 0.14112
4 -0.7568025

```

### 15.1.3 by B

kSQL also has a way to group rows using `by`. The result is a table where the key is the now unique field in the `by`.

```

n:5;t:[[]x:!n;y:sin !n]

select sum y by x>2 from t
x	y
0|1.750768
1|-0.6156825

```

### 15.1.4 where C

kSQL makes it easy to build up a where clause to filter down the table. C is the list of constraints.

```

n:5;t:[[]x:!n;y:sin !n]
select from t where x>0
x y
- -----
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025

select from t where (x>0)&y within 0 .9
x y
- -----
1 0.841471
3 0.14112

```

### 15.1.5 Query with By and Where

```

n:5;t:[[]x:!n;y:sin !n]
select sum y by x>2 from t where y>0
x	y
0|1.750768
1|0.14112

```

### 15.1.6 update

`update` allows one to modify values without specifying all the fields that pass through. `update` is also used to add new columns in a table. `update` does not save the table that requires an assignment.

```

t:[[]x:`a`b`c;y:1 2 3];t
x y

```

```

- -
a 1
b 2
c 3

 update y+18 from t
x y
- --
a 19
b 20
c 21

 update z:y+18 from t
x y z
- - --
a 1 19
b 2 20
c 3 21

 t:update z:y+18 from t;t
x y z
- - --
a 1 19
b 2 20
c 3 21

```

### 15.1.7 delete

Delete rows from a table that satisfy one or more conditions. Currently not working.

```

t:[[]x:`a`b`c;y:1 2 1];t
x y
- -
a 1
b 2
c 1

 delete from t where y>1
!value

 delete from t where x=`c,y=1
!nyi

```

## 15.2 Joins

k9 has a number of methods to join tables together which are described below. In this section t, t1 and t2 represent tables and k, k1 and k2 represent utables.

### join syntax

[union],t2  
 page 75,  
 [leftjoin],  
 page 75,  
 [outer],k2  
 page 76,  
 [asof],k  
 page 76,  
 [asof],+k2  
 page 77,

### 15.2.1 union join $\Rightarrow$ t1,t2

Union join table t1 with table t2. The tables should have the same columns and the join results in a table with t2 appended to t1. If the tables do not have the same columns then return t1.

```
t1:[[]s:`a`b;p:1 2;q:3 4]
t2:[[]s:`b`c;p:11 12;q:21 22]
```

```
t1
s p q
- - -
a 1 3
b 2 4
```

```
t2
s p q
- - -
b 11 21
c 12 22
```

```
t1,t2
s p q
- - -
a 1 3
b 2 4
b 11 21
c 12 22
```

### 15.2.2 leftjoin $\Rightarrow$ t,k

leftjoin table t with utable k. Result includes all rows from t and values from t where there is no k value.

```
t:[[]s:`a`b`c;p:1 2 3;q:7 8 9]
k:[[]s:`a`b`x`y`z;q:101 102 103 104 105;r:51 52 53 54 55]
```

```
t
s p q
- - -
a 1 7
```

```

b 2 8
c 3 9

```

```

 k
s|q r
-|--- --
a|101 51
b|102 52
x|103 53
y|104 54
z|105 55

```

```

 t,k
s p q r
- - --- --
a 1 101 51
b 2 102 52
c 3 9 0

```

### 15.2.3 outer join $\Rightarrow$ k1,k2

Outer join utable k1 with key utable k2.

```

k1:[[s:`a`b]p:1 2;q:3 4]
k2:[[s:`b`c]p:9 8;q:7 6]
k1
s|p q
-|- -
a|1 3
b|2 4

```

```

k2
s|p q
-|- -
b|9 7
c|8 6

```

```

k1,k2
s|p q
-|- -
a|1 3
b|9 7
c|8 6

```

### 15.2.4 asof join $\Rightarrow$ t,k

Asof joins a table t to a utable k (key by time) such that the t values show the preceeding or equal time value of k.

```

t:[[]t:09:30+5*!5;p:100+!5];t

```

```

t p
----- ---
09:30 100
09:35 101
09:40 102
09:45 103
09:50 104

k:[t:09:32 09:41 09:45]q:50 51 52];k
t	q
09:32|50
09:41|51
09:45|52

t,k
t p q
----- --
09:30 100 0
09:35 101 50
09:40 102 50
09:45 103 52
09:50 104 52

```

Scaling this up to a bigger set of tables one can see the performance of k9 on joins.

```

N:_1e8;T:[[]t:N?`t 0;q:N?100];5#T
t q
----- --
00:00:00.001 44
00:00:00.002 46
00:00:00.002 48
00:00:00.003 35
00:00:00.003 43

n:_1e5;K:[t:n?`t 0]p:n?100];5#K
t	p
00:00:00.481|54
00:00:00.961|63
00:00:01.094|67
00:00:01.479|16
00:00:01.917|58

\t T,K
222

```

### 15.2.5 asof+ join $\Rightarrow$ k1+k2

Asof+ joins utables adding the non-key fields to represent the sum asof the key field usually time. This join allows one to aggregate over markets to find the total available at a given time. The utables need to be specified with `a.

```
k1:`a [[t:09:30+5*!5]bs:100*1 2 3 2 1];k1
t	bs
09:30|100
09:35|200
09:40|300
09:45|200
09:50|100

k2:`a [[t:09:32 09:41 09:45]bs:1 2 3];k2
t	bs
09:32| 1
09:41| 2
09:45| 3

k1+k2
t	bs
09:30|100
09:32|101
09:35|201
09:40|301
09:41|302
09:45|203
09:50|103
```

## 15.3 Insert and Upsert

One can add data to tables via insert or upsert. The difference between the two is that insert adds data to a table while upsert will add or replace data to a keyed table. Upsert adds when the key isn't present and replaces when the key is.

### 15.3.1 insert $\Rightarrow$ t,d

Insert dictionary d into table t.

```
t:[[]c1:`a`b`a;c2:1 2 7];t
c1 c2
-- --
a 1
b 2
a 7
```



```

t,`c1`c2!(`a;12)
c1 c2
-- --
a 1
b 2
a 7
a 12

```

```

t,`c1`c2!(`c;12)
c1 c2
-- --
a 1
b 2
a 7
c 12

```

### 15.3.2 upsert $\Rightarrow$ k,d

Insert dictionary d into utable k.

```

k:[c1:`a`b`c]c2:1 2 7];k
c1	c2
a | 1
b | 2
c | 7

```

```

k,`c1`c2!(`a;12)
c1	c2
a |12
b | 2
c | 7

```

```

k,`c1`c2!(`b;12)
c1	c2
a | 1
b |12
c | 7

```

## 16 System

k9 comes with a few system functions and measurement commands. The commands allow you to load a script, change the working directory, measure execution times and memory usage, and list defined variables.

```
System
\l a.k [load], page 80
\t:n x [timing], page 80
\u:n
\v [variables], page 80
\w [memory], page 80
\cd x [cd], page 80
```

### 16.1 load $\Rightarrow$ \l a.k

Load a text file of k9 commands. The file name must end in .k.

```
\l func.k
\l func.k
\l func.k7 / will error as not .k
:nyi
```

### 16.2 timing $\Rightarrow$ \t

List time elapsed

```
\t ^(_1e7)?_1e8
227
```

### 16.3 variables $\Rightarrow$ \v

List variables

```
a:1;b:2;c:3
\v
[v:`a`b`c]
```

### 16.4 memory $\Rightarrow$ \w

List memory usage

```
\w
0
r:(`i$10e6)?10
\v
2097158
```

### 16.5 cd $\Rightarrow$ \cd x

Change directory (cd) into x

```
\cd scripts
```

## 17 Control Flow

Although not heavily used in k9, control flow statements will still be required occasionally.

`$(b;t;f)` Control

### 17.1 `cond` $\Rightarrow$ `$(b;t;f)`

If `b` is non zero then `x` else `y`. `x` and `y` are not required to be the same type.

```
$(3>2;`a;`b)
`a
```

```
$(2>3;`a;`b)
`b
```

```
$(37;12;10)
12
```

```
$(1b;`a`b!(1 2;3 4);`n)
a|1 2
b|3 4
```

```
$(1b;a:3;b:2)
3
a
3
b / is not set as the f case not evaluated
!value
```

## 18 Temporal Functions

k9 has functions (within `.z`) to get the current date and time in a number of precisions. There are also functions to retrieve partial date and time using dot notation.

```
date 2001.01.01 .z.d
time 12:34:56.789 .z.t
```

### 18.1 `.z.[dm]` date

These functions retrieve the date as day (d) or month (m) precision.

```
.z.d
2020.12.13
```

```
.z.m
2020.12.
```

### 18.2 `.z.[hrstuv]` time

These functions retrieve the time as hour (h), minute (r), second (s), millisecond (t), microsecond (u), or nanosecond (v) precision.

```
.z.h
11
.z.r
11:25
.z.s
11:25:16
.z.t
11:25:16.842
.z.u
11:25:17.491362
.z.v
11:25:18.186360064
```

One could use the current time commands to measure run time but typically this is done via `\t`

```
t1:.z.v;(_2e8)?1.;t2:.z.v;t2-t1
00:00:00.609207008

\t (_2e8)?1.
610
```

### 18.3 Temporal dot functions

These functions use dot notation to retrieve partial date and times.

```
now:.z.v;now
11:27:18.049558016
now.h
```

```
11
 now.r
11:27
 now.s
11:27:18
 now.t
11:27:18.049
 now.u
11:27:18.049558
 now.v
11:27:18.049558016
```

## 19 Errors

Given the terse syntax of k9 it likely won't be a surprise to a new user that the error messages are rather short also. The errors are listed on the help page and described in more detail below.

```
error: [class], page 84, [rank], page 84, [length], page 84, [_type], page 84, [domain],
page 84, [limit], page 84, stack [parse], page 85, [_value], page 85
```

### 19.1 :class

Calling a function on mismatched types.

```
3+`b
:class
```

### 19.2 :rank

Calling a function with too many parameters.

```
{x+y}[1;2;3]
{x+y}[1;2;3]
^
:rank
```

### 19.3 :length

Operations on unequal length lists that require equal length.

```
(1 2 3)+(4 5)
:length
```

### 19.4 :type

Calling a function with an unsupported variable type.

```
`a+`b
^
:type
```

### 19.5 :domain

Exhausted the number of input values

```
-12?10 / only 10 unique value exist
:domain
```

### 19.6 :limit

Exceeded a limit above the software maximum, eg. writing a single file above 1GB.

```
n:_100e6;d:+`x`y!(!n;n?1.);`d 2:d
n:_100e6;d:+`x`y!(!n;n?1.);`d 2:d

:limit
```

## 19.7 :nyi

Running code that is not yet implemented. This may come from running code in this document with a different version of k9.

```
2020.05.31 (c) shakti
 =+`a`b!(1 2;1 3)
a b|
- -|-
1 1|0
2 3|1
```

Aug 6 2020 16GB (c) shakti

```
 =+`a`b!(1 2;1 3)
 =+`a`b!(1 2;1 3)
 ^
 :nyi
```

## 19.8 :parse

Syntax is wrong. Possible you failed to match characters which must match, eg. (), {}, [], "".

```
 {37 . "hello"
 :parse
```

## 19.9 :value

Undefined variable is used.

```
 g / assuming 'g' has not be defining in this session
 :value
```

## 20 Conclusion

I expect you are surprised by the performance possible by k9 and the fact that it all fits into a 134,152 bytes! (For comparison the ls program weighs in 51,888 bytes and can't even change directory.)

If you're frustrated by the syntax or terse errors, then you're not alone. I'd expect most had the same problems, persevered, and finally came away a power user able to squeeze information from data faster than previously imagined.

If you've come along enough to get it, then you'll probably realised this manual isn't needed and it's all here...

| Verb |         | Adverb  | Type                                  | System                    |
|------|---------|---------|---------------------------------------|---------------------------|
| :    | x       | y       | f/ over c/ +join char " ab"           | \l a.k                    |
| +    | flip    | plus    | f\ scan c\ +split name ``ab           | \t:n x                    |
| -    | negate  | minus   | f' each v' +has int 0 2 3             | \u:n x                    |
| *    | first   | times   | f': eachp v': +bin flt 0 2 3.         | \v                        |
| %    |         | divide  | f/: eachr ([n;]f)/: date 2001.01.01   | .z.d                      |
| &    | where   | min/and | f\: eachl ([n;]f)\: time 12:34:56.789 | .z.t                      |
|      | reverse | max/or  |                                       |                           |
| <    | asc     | less    |                                       |                           |
| >    | dsc     | more    |                                       |                           |
| =    | group   | equal   | I/O                                   |                           |
| ~    | not     | match   | 0: r/w line                           | Class \f                  |
| !    | enum    | key     | 1: r/w byte                           | List (2;3.4;`c) \ft x     |
| ,    | enlist  | cat     | *2: r/w data                          | Dict [a:2;b:`c] \fl x     |
| ^    | sort    | [f]cut  | *3: k-ipc set                         | Func {[a;b]a+b} \fc x     |
| #    | count   | [f]take | *4: https get                         | Expr :a+b \cd [d]         |
| -    | floor   | [f]drop |                                       |                           |
| \$   | string  | cast    | [\$[b;t;f] Control                    | Table                     |
| ?    | unique+ | find+   |                                       | t:[[]i:2 3;f:3 4.;s:`a`b] |
| @    | type    | [f]at   | @[r;i;f[;y]] amend                    | utable [[b:..]a:..]       |
| .    | value   | [f]dot  | .[r;i;f[;y]] dmend                    | xtable `...![[a:..]       |
| \\   | exit    |         | / comment                             | [if do while]             |

```
select[C]{count first last sum min max *[avg var dev med ...]}from T where B
update C from T where B; delete from T where B; key unkey meta in bin within
exp log sin cos sqr sqrt div mod bar; top freq msum mavg sums deltas prm cmb
```