Shakti tutorial

John Estrada

This manual is for Shakti (k9) build 2020.09.29.
3 October 2020

1 Intro

The k9 programming language is designed primarily for the analysis of data. It may surprise new users with two rather different paradigms, (1) extremely fast data analysis and (2) an extremely concise syntax. After some familiarity these changes 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, 40 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 to memory and compute a statistic, average difference over each table, which uses each and every row.

```
 bash-3.2$ 2020.09.29 17GB 4core (c) shakti
 \t q:2:`q;{select s:avg a-b from x}'q[!q]
415
```

That's 415ms to read the data in from disk and compute over all the 40 million values. The data read is the biggest bit. If the data was already in memory then it's even faster.

```
 \t {select s:avg a-b from x}'q[!q]
105
```

105ms, not bad.

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

```
nf:d+*|d:(|-d),d:683 954 997 1000;
T:{`t'_+\(-1+86.4e6)*x%+/x}?[;1.]@
B:100++\1e-2*-3+nf bin/:?[;*|nf]@
S:?[;1e-2*2,2,8#1]@
q:`eurusd`usdjpy`usdchf!@{select t,b,a:b+s from +`t`b`s!(T;B;S)@'x@}'_20e6 10e6 10e6
`q 2:q
```

1.2 Going concise

The k9 language is more closely related to mathematics syntax than most programming lanauges. It requires the developer to learn to speak k9 but once that happens most find an ability to "speak" quicker 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 trival operations like arithmetic most programming languages use symbols also. Moving on to something less math like most programming lanauges switch to clear words while k9 remains with symbols which turn out to have the same level of clarity. As an example, to determine the distinct values of a list most programming languages might use a synatx 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, harder to bug, and easier to maintain.

In math 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 do you 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 likely appreciate symbols are shorter but look like line noise. That's true but so did arithetic until you learned the basics.

When you first learned arithmetic you likley didn't have a choice. Now you have a choice about learning k9. If you give it a try, then I expect 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 it's file permissions to allow it to execute.

```
chomd 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 the terminal gives you a concise overview of the language. This document aims to provide details to beginning users where the help screen is a tad too terse. Some commands are not yet complete and thus marked with an asterisk, eg. *?[x;i;f[;y]] splice.

```
2020.09.29 17GB 4core (c) shakti
 \
Verb                      Adverb               Noun              System
:    x         y          f/  over  c/  join   bool 011b         \l a.k
+    flip      plus       f\  scan  c\  split  int  0N 0 2 3     \t:n x
-    negate    minus      f'  each  v'  has    flt  0n 0 2 3.4   \u:n x
*    first     times      f': eachp v': bin    char " ab"        \v
%              divide     f/: eachr ([n;]f)/:  name ``ab         \w
```

```
&    where     min/and    f\: eachl ([n;]f)\:  uuid              \cd [x]
|    reverse   max/or                .z.dtv time  12:34:56.123456789
<    asc       less                  .z.DTV date 2024.01.01T12:34:56
>    dsc       more
=    group     equal      I/O
~    not       match      0:  r/w line         Class
!    enum      key        1:  r/w char         List (2;3.4;`a)
,    enlist    cat       *2:  r/w data         Dict `i`f!(2;3.4)
^    sort   [f]cut       *3:  kipc set         Func {[a;b]a+b}
#    count  [f]take      *4:  http get         Expr :a+b
_    floor  [f]drop
$    string    cast+      $[b;x;y] if else                       \f
?    unique+   find+      ?[x;i;f[;y]] splice  Table [[]a:..;..] \ft x
@    type   [f]at         @[x;i;f[;y]] amend   XTab  `a..![[]..] \fc x
.    value  [f]dot        .[x;i;f[;y]] dmend   TTab   [[a:..]..] \fl x
sqrt sqr exp log sin cos div mod bar in bin within
count first last min max sum avg
[select|*update] A by B from T where C ; *delete from T where C
t:[[]i:2 3;f:2.3 3.4;s:`ab`abc]; `json t ; t~`csv?`csv t
\\ exit      / comment   #[if do while select update]

*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 ..
python: import k;      k.k('+',2,3)
nodejs: k=require('k');k.k('+',2,3)

error: class rank length type domain value (parse limit stack)
limit: {[param8]local8 global32 const128 jump256} name256

*production: no limits -- 10 times faster than trial (1GB 1core)
```

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 generally a useful option to have.

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 be 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` 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

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
 {[x](+/m*m:x-avg x)%#x}'+q  / calculate the variance of each symbol
a|0.6601538
b|0.6629631
c|0.6708467
```

1.7 Document formatting for code examples

This document uses a number of examples to help clarify k9. The sytax is that input has a leading space and output does not. 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 nuances

One will need to understand some basic rules of k9 in order to progress. These will likely seem strange at first but the faster you learn a few nuances the faster you'll move forward.

1.8.1 The language changes often.

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

1.8.2 : 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 % is used to divide numbers

Yeah, 2 divide by 5 is written as `2%5` and 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

`+` does not happen specially before or after *. 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.

```
 *(3;6;9)    / single argument so * is first element of the list
3
 2*(3;6;9)   / two arguments so * is multiplication
6 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.

If this book wasn't a simples guide then lists (l) and functions (f) would be replaced by maps (m) given the interchangeability. One way to determine if a map is either a list or function is via the type function. Lists and functions do not have the same type.

```
 l:3 4 7 12
 f:{[x]3+x*x}
 l@2
7
 f@2
7
```

1.8.8 k9 is expressed in terms of grammar.

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 appple. (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 >' 0 1 2 3 4 5 (Noun verb adverb noun)`

2 Examples

Before jumping into synax let's look at some example problems to get a sense of the speed of k9 at processing data. Given both the historic use of languages similar to k9 in finance and the author's background much of the examples will be based on financial markets. For those not familiar with this field a short introduction will likley be needed.

2.1 A Tiny Introduction to Financial Market Data

Financial market data generally are stored as prices and trades. Prices will include at a minimum time, securtiy, price to buy and price to sell. Trades will include at a minimum time, security, and trade price. In normal markets there are many more prices than trades.

Let's use k9 to generate a set of random prices for a single security hence elimating the need for that field.

```
 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 and that's because it apparently took you 1s 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 financial data and compute basic statistics quickly. This table takes about 4 GB and 3.3 seconds on a relatively new consumer laptop.

```
 n:_100e6                              / 100 million rows
 t:{[x]09:00:00.000+x?10:00:00.000}     / random times
 s:{[x]x?`a`b`c`d`e}                   / random symbols
 m:0,(|m),365378984,m:271810244 42800467 2636454 62769 572 2;
 d:{[x](-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 min t, max t from q        / min and max time values
t|09:00:00.000
t|18:59:59.999

 select #s by s from q             / count each symbol
s|s
-|--------
a|19999325
b|20000982
c|19996938
d|20001721
e|20001034

 select #d by d from q             / check the normal distribution
d |d
--|--------
-6|       1
-5|      46
-4|    6284
-3|  263124
-2| 4276881
-1|27184896
 0|36538226
 1|27182073
 2| 4278498
 3|  263523
 4|    6391
 5|      57
```

2.3 Understanding Code Examples

In the shakti mailing list there is a number of code examples that can be used to learn best practice. In order to make sense of other's codes one needs to be able to effeciently parse the typically dense k9 language. Here, an example of how one goes about this process is presented.

```
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.

```
 000000000011111111112222222222333333
 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 k9 evaluates right to left let’s start with the right most 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 understand 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 only use 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 charcters 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 add an integer list as long as the send argument to each value. In order to ensure this is clear one could write something similar and ensure the output is able to be predicted.

```
 {(!x)+\!y}[6;4]
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
```

Now using the matrix above the code indices the first argument and pull substrings that match in length of the search string.

```
 {x@(!#x)+\!#y}["Find the +++ needle in + the ++ text";"++"]
Fi
in
nd
d
 t
th
he
e
 +
++
++
+
```

```
 n
ne
ee
ed
dl
le
e
 i
in
..
```

At this point one can compare the search substring in this list of substrings to find a match.

```
 {(x@(!#x)+\!#y)~\y}["Find the +++ needle in + the ++ text";"++"]
0000000001100000000000000000001000000b
```

And then one can use the where function, `&`, to determine the index of the matches.

```
 {&(x@(!#x)+\!#y)~\y}["Find the +++ needle in + the ++ text";"++"]
9 10 29
```

3 Benchmarks

Shakti seems likely to be one of the faster data analysis languages out there and clear benchmarks always help to illuminate the matter. The Shakti website has a file for such purpose, b.k. You can see below for the first query (Q1) k9 takes 1ms while postgres, spark and mongo are orders of magntitude 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:?[;"ABCDEFGHIJ"]
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]
"AADCBCCCBC"
```

3.1.3 m, n, S, N

m is the number of symbols. n is the number of trades. S is the list of symbol names. N is a list of decreasing numbers which sum approximately to n. (Approximately as the values are ceil to integers).

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

3.1.4 t

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

Pulling 1 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 XTab of quotes. The fields are time (t), exchange (e), and bid (b). The number of quotes is set by 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 is the first 100 symbols of S.

```
 a
`PKEM
```

3.1.7 Max price by exchange

The query takes 100 tables from the trade XTab 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 XTab 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 XTab 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 XTabs, 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 Verb

This chapter covers verbs which are the corefunctions of k9. Given how different is it to call functions in k9 than many other languages this is probably a chapter that will have to be covered 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 and type of arguments. This reuse of symbols is also an item that causes confusion for 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!)

4.1 x ⇒ :x

4.2 set ⇒ x:y

Set a variable, x, to a value, y.

```
 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 ⇒ +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 5
 +`a`b!+x
a b
- -
1 2
3 4
5 6
```

4.4 plus ⇒ x+y

Add x and y.

```
 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, will error :length
: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 ⇒ -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 ⇒ x-y.

Subtract y from x.

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

4.7 first ⇒ *x

Return the first value of x. Last can either be determine by taking the first element of the reverse list (`*|`‘a‘b‘c) or using last syntax ((:/)‘a‘b‘c).

```
 *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
```

4.8 times ⇒ 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 ⇒ x%y

Divide x by y.

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

4.10 where ⇒ &x

Given a list of 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
 &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
```

4.11 and ⇒ 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 ⇒ |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
```

4.13 or ⇒ 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) ⇒ < (>) x

The indices of a list in order to sort the list in ascending (descending) order.

```
 <2 3 0 12
2 0 1 3
 x@<x:2 3 0 12
0 2 3 12
```

4.15 less (more) ⇒ x < (>) y

x less (more) than y.

```
 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 ⇒ =x

A dictionary of the disinct values of x (key) and 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 ⇒ x=y

x equal to y

```
 2=2
1b
 2=3
0b
 "banana"="abnaoo"
001100b
```

4.18 not ⇒ ~x

Boolean invert of x

```
 ~1b
0b
 ~101b
010b
 ~37 0 12
010b
```

4.19 match ⇒ x~y

Compare x and y.

```
 2~2
1b
 2~3
0b
 `a`b~`a`b  / different than = which is element-wise comparison
1b
 `a`b=`a`b
11b
```

4.20 enum ⇒ !x

Given an integer, x, generate an integer list from 0 to x-1.

```
 !3
0 1 2
```

Given a list of integers, x, generate a list of lists where each individual index goes from 0 to n-1. Aka an odometer where the each place can have a separate base and the total number of lists is the product of all the x values.

```
 !2 8 16  / an odometer where the values are 0-1, 0-7, and 0-15.
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 1 1 1 1 1 1 1 1 1  1  1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11
```

```
 12#+!2 8 16 / flip the output and display first 18 rows
0 0  0
0 0  1
0 0  2
0 0  3
0 0  4
0 0  5
0 0  6
0 0  7
0 0  8
0 0  9
0 0 10
0 0 11
 5_+!2 8 16 / flip the output and display last 5 rows
1 7 11
1 7 12
1 7 13
1 7 14
1 7 15
 B:`b$+!16#2 / create a list of 16-bit binary numbers from 0 to 65535
 B[12123]    / pull the element 12,123
0010111101011011b
 2/:B[12123] / convert to base10 to check it's actually 12123
12123
```

4.21 key ⇒ x!y

Dictionary of x (key) and y (value). If looking to key a table then refer to [[f]cut], page 23.

```
 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 ⇒ ,x

Create a list from x

```
 ,3
,3
 ,1 2 3
1 2 3
 3=,3
,1b
 3~,3
```

```
0b
```

4.23 cat ⇒ x,y

Concatenate x and y.

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

4.24 sort ⇒ ^x

Sort list x into ascending order.

```
 ^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 ⇒ x^y

Cut list y by size or indices x. Also, cut y into a domain (x) and range.

Cut list.

```
 3^!18
 0  1  2
 3  4  5
 6  7  8
 9 10 11
12 13 14
15 16 17
 0 1 5^0 1 2 3 4 5 6 7 8 9
0
1 2 3 4
5 6 7 8 9
 1 5^0 1 2 3 4 5 6 7 8 9
1 2 3 4
5 6 7 8 9
```

Cut into domain and range, aka key a table.

```
 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 ⇒ #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 ⇒ x#y

Take is used to return a subset of list y depending if x is a atom, list, or function. If x is an atom, then postive (negative) x returns the first (last) x elements of y. If x is a list, then returns any values common in both x and y. If x is a function (f), then filter out values where the funtion is non-zero.

```
 3#0 1 2 3 4 5          / take first
0 1 2
 -3#0 1 2 3 4 5         / take last
3 4 5
 2#"hello"
"he"
 (1 2 3 7 8 9)#(2 8 20) / common
2 8
 (0.5<)#10?1.           / filter
0.840732 0.5330717 0.7539563 0.643315 0.6993048f
```

4.28 floor ⇒ _x

Return the integer floor of float x.

```
 _3.7
3
```

4.29 [f]drop ⇒ x_y

Drop is used to remove a subset of list y depending if x is a atom, list, or function. If x is an atom, then postive (negative) x removes the first (last) x elements of y. If x is a list, then remove any values in x from y. If x is a function (f), then remove values where the funtion is non-zero.

```
 3_0 1 2 3 4 5          / drop first
3 4 5
 -3_0 1 2 3 4 5         / drop last
0 1 2
 2#"hello"
"he"
 (1 2 3 7 8 9)_(2 8 20) / drop common
,20
 (0.5<)_10?1.           / drop true
0.4004211 0.2929524f
```

4.30 string ⇒ $x

Cast x to string.

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

4.31 cast+ ⇒ 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
```

Multiple matrices x and y together.

```
 (0 1 2;3 4 5;6 7 8)$(10 11;20 21;30 31)
 80  83
260 272
440 461
```

4.32 unique+ ⇒ ?x

Return the unique values of the list x. The `?` preceeding the return value explicitly shows that list has no repeat values.

```
 ?`f`a`b`c`a`b`d`e`a
```

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

4.33 find+ ⇒ x?y

Find the first element of x that matches y otherwise return the end of vector. Also, acts to generates random numbers from 0 to y when x and y are integers.

```
 `a`b`a`c`b`a`a?`b
1
 `a`b`a`c`b`a`a?`d
7
 0 1 2 3 4?10
5
 (1;`a;"blue";7.4)?3
4
 3?10         / 3 random integers between 0 and 9 inclusive
5 5 6
 3?10         / as above but no repeats
0 5 6
```

4.34 type ⇒ @x

Return the data type of x.

```
 @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 ⇒ 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
```

4.36 value ⇒ .x

Value a string of valid k code or list of k code.

```
 ."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

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

4.37 [f]dot ⇒ 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]].

action	**@y**	**#y**	**example**
simple index	‘I	1	,2
simple indices	‘I	1	,1 3

recursive index	‘L	1	0 2
recursive index over	‘L	2	(0 2;1 3)

```
 (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
```

5 Adverb

Adverbs modify verbs to operative iteritvely over nouns. This previous sentence likley will only make sense 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`.

```
Adverb
f/  [scan], page 29,  c/  [join], page 29
f\  [scan], page 29,  c\  [split], page 29
f'  [each], page 30,  v'  [has], page 30
f': [eachp], page 30, v': [bin2], page 30.
f/: [eachr], page 31, ([n;]f)/:
f\: [eachl], page 31, ([n;]f)\:
```

5.1 scan (over) ⇒ 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 → x@0
- f@1 → f[f@0;x@1]
- ...
- f@i → f[f@i-1;x@i]
- ...
- f@n → f[f@n-1;x@n]

An example

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

5.2 join ⇒ c/x

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

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

5.3 split ⇒ c\x

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

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

5.4 each ⇒ f'x

Apply function f to each value in list x.

```
 *((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 ⇒ v'x

Determine if vector v has element x. Simliar to [in], page 47, but arguments reversed.

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

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

5.6 eachp ⇒ 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
```

5.6.1 bin ⇒ 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 ⇒ f/:

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

```
 (!2)+/:!4
0 1
1 2
2 3
3 4

 +/:[!2;!4]
0 1
1 2
2 3
3 4
```

5.8 eachl ⇒ f\[x;y]

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

```
 (!2)+\:!4
0 1 2 3
1 2 3 4

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

5.9 n scan (n over) ⇒ x f\:y (x f/:y)

Compute f with initial value x and over list y. f[i] = f[f[i-1];y[i]] except for the case of f[0]=f[x;y[0]]. n over differs from n scan in that it only returns the last value.

```
 f:{(0.1*x)+0.9*y}      / ema
 0. f\:1+!3
0.9 1.89 2.889
 f:{[x;y](`$,/$x),(`$,/$y)} / join and collapse
 `x f\: `y0`y1`y2
x     y0
xy0   y1
xy0y1 y2
 `x f/: `y0`y1`y2
xy0y1 y2
```

5.10 c(onverge) scan ⇒ f\:x

Compute f[x], f[f[x]] and continue to call f[previous result] until the output converges to a stationary value or the output produces x.

```
  {x*x}\:.99
0.99 0.9801 0.960596 0.9227447 .. 9.420123e-144 8.873872e-287 0
```

5.11 c(onverge) over ⇒ f/:x

Same as converge scan but only return last value.

5.12 vs ⇒ x\:y

Convert y (base 10) into base x.

```
 2\:129
10000001b
 16\:255
15 15
```

5.13 sv ⇒ x/:y

Convert list y (base x) into base 10.

```
 2/:10101b
21
 16/:15 0 15
3855
```

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 additional 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 seperate chapter.

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

```
Atom   Example
[bool], page 33,  011b
[int], page 33,   0N 2 3
[flt], page 34,   0n 2 3.4
[char], page 34,  " ab"
[name], page 34,  ``ab
[uuid], page 35,
[time], page 35,  12:34:56.123456789
[date], page 35, 2024.01.01T12:34:56
```

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 evalue 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 bool ⇒ 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 Numeric Data

Numbers can be stored as integers and floats.

6.2.1 int ⇒ Integer i

Integers

```
 3
3
 3+1
4
 @3
`i
 a:3;
 @a
`i
 3%1   / result will be float even though inputs are int
3f
```

6.2.2 flt ⇒ Float f

Float

```
 3.1
3.1
 3.1+1.2
4.3
 3.1-1.1
2f
 @3.1-1.1
`f
 @3.1
`f
 a:3.1;
 @a
`f
```

6.3 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.

6.3.1 char ⇒ 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
```

6.3.2 name ⇒ Name n

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

```
 @`blue
```

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

6.4 Unique Identifier

TBD

6.4.1 uuid ⇒ Uuid

TBD

6.5 Temporal Data

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

6.5.1 time ⇒ Time t

Times are stored in hh:mm:ss.123 format and stored internally as integers.

```
 @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
```

6.5.2 date ⇒ Date D

Dates are in yyyy.mm.dd format and stored internally as integers.

```
 @2020.04.20               / date
`D
 .z.D                       / current date in GMT
2020.09.13
 `i .z.D                    / numeric representation of date
-1205
 `i 2024.01.01              / zero date
0
 `D 0                       / zero date
2024.01.01
```

6.5.3 datetime ⇒ Datetime S

Dates and times can be combined as 2020.04.20T12:34:56.789.

```
 @2020.04.20T12:34:56.789  / date and time
`T
 "T"$"2020.04.20 12:34:56.789" / converting from string
2020-04-20T12:34:56.789
 "T"$"2020-04-20 12:34:56.789"
2020-04-20T12:34:56.789
 "T"$"2020.04.20T12:34:56.789"
2020-04-20T12:34:56.789
 "T"$"2020-04-20T12:34:56.789"
2020-04-20T12:34:56.789
```

6.6 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

k9 is optimized for operations on uniform lists of data. In order to take full advantage one should store data in lists or derivatives of lists, eg. dictionaries or tables, and operate on them without explicit iteration.

7.1 List Syntax

In general, lists are created by data separated by semicolons and encased by parenthesis. Uniform lists can use a simpler syntax of spaces between elements.

```
 a:1 2 3
 b:(1;2;3)
 a~b             / are a and b the same?
1b
 @a             / uniform lists are upper case value an element
`I
 @'a            / type of each element
`i`i`i
 c:(1i;2f;"c";`d)
 @c             / nonuniform lists are type `L
`L
 @'c
`i`f`c`s
 c:1i 2f "c" `d  / incorrect syntax for nonuniform list
:type
```

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 likley more clear.

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

7.3 Updating List Elements

Lists can be updated element wise but typically one is likely to be updating many elements and there is a syntax for doing so.

```
 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
 a:@[a;0 2 4 6 8;0];a
0 4 0 80 0 12 0 16 0 20
 a:@[a;1 3 5;*;100];a
0 400 0 8000 0 1200 0 16 0 20
 a:@[a;!#a;:;0];a
```

List amend syntax has a few options so will be explained in more detail.

- @[list;indices;value]
- @[list;indices;identity function;value]
- @[list;indices;function;value]

The first syntax sets the list at the indices to value. The second syntax performs the same modificaiton but explicitly lists the identity function, :. The third synatx is the same as the preceeding but uses an arbitrary fuction.

Often the developer will need to determine which indices to modify and in cases where this isn't onerous it can be done in the function.

```
 a:2*1+!10
 @[a;&a<14;:;-3]
-3 -3 -3 -3 -3 -3 14 16 18 20
 @[!10;1 3 5;:;10 20 30]
0 10 2 20 4 30 6 7 8 9
 @[!10;1 3 5;:;10 20]   / index and value array length mistmatch
:length
 @[!10;1 3;:;10 20 30]  / index and value array length mistmatch
:length
```

7.4 Fuction 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 function only operates on x[i] and y[i] 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 : Take`

```
 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
10
8 8
6 6 6
4 4 4 4
2 2 2 2 2
```

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 : Unique values to 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. 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 or tables.

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 ⇒ x!y

```
 d0:`pi`e`c!3.14 2.72 3e8;d0            / elements
pi|3.14
e |2.72
c |3e+08

 d1:`time`temp!(12:00 12:01 12:10;25.0 25.1 25.6);d1 / lists
time|12:00 12:01 12:10
temp|25 25.1 25.6

 d2:0 10 1!37.4 46.3 0.1;d2             / keys as numbers
 0|37.4
10|46.3
 1|0.1

 d3:`a`b`a!1 2 3;d3                     / non-unique keys
a|1
b|2
a|3

 d3`a                                   / `a value returned
1
```

8.2 Dictionary Indexing ⇒ x@y

Dictionary indexing, 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 notaions for indexing work, output surpressed
```

```
 x@`b;  / at
 x(`b); / parenthesis
 x `b;  / space
 x[`b]; / square bracket
```

8.3 Dictionary Key ⇒ !x

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

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

8.4 Dictionary Value ⇒ x[]

The values from a dictionary are retreived by bracket notation.

```
 d0[]
pi   e    c
3.14 2.72 3e+08

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

 d2[]
0    10   1
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]
25f
```

8.5 Sorting a Dictionary by Key ⇒ ^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 ⇒ <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 ⇒ +x

This command flips a dictionary into a table but will be covered in detail in the table section. Flipping a dictionary whose values are a single element has no effect.

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

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

 do~+d0
1b

 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 repeate 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 dictions that operate over the values. If 'f' is a function 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 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.

math: [sqrt], page 46, [exp], page 46, [log], page 46, [sin], page 46, [cos], page 46, [div], page 46, [mod], page 46, [bar], page 47
wrapper: count first last min max sum avg
range: [in], page 47, [bin], page 47, [within], page 47

9.1 Math Functions ⇒ sqrt exp log sin cos div mod bar

9.1.1 sqrt ⇒ sqrt x

```
 sqrt 2
1.414214
```

9.1.2 exp ⇒ exp x

```
 exp 1
2.718282
```

9.1.3 log ⇒ log x

Log computes the natural log.

```
 log 10
2.302585
```

9.1.4 sin ⇒ sin x

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

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

9.1.5 cos ⇒ cos x

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

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

9.1.6 div ⇒ x div y

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

```
 2 div 7
3
 5 div 22
4
```

9.1.7 mod ⇒ 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
```

9.1.8 bar ⇒ 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
```

9.2 Wrapper Functions ⇒ count first last min max sum avg

These functions exist as verbs but also can be called with the names above.

```
 n:3.2 1.7 5.6
 sum n
10.5
 +/n
10.5
```

9.3 Range Functions ⇒ in bin within

9.3.1 in ⇒ x in y

Determine if x is in list y. Similar to [has], page 30, but arguments reversed.

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

9.3.2 bin ⇒ 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
```

9.3.3 within ⇒ x within y

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

```
 3 within (0;12)
1b
 12 within (0;12)
0b
 23 within (0;12)
```

0b

10 More Functions

10.1 cond ⇒ $[x;y;z]

If x then y else z.

```
 $[3>2;`a;`b]
`a
 $[2>3;`a;`b]
`b
```

10.2 amend ⇒ @[x;i;f[;y]]

Replace the values in list x at indices i with f or f[y].

@[x;i;f] examples

```
 x:(`x00`x01;`x10`x11`x12;`x20;`x30`x31`x32);x
x00 x01
x10 x11 x12
x20
x30 x31 x32

 @[x;,1;`newValue]
x00 x01
newValue
x20
x30 x31 x32

 @[x;1 2;`newValue]
x00 x01
newValue
newValue
x30 x31 x32
```

@[x;i;f;y] examples

```
 x:(0 1;10 11 12;20;30 31 32);x
0 1
10 11 12
20
30 31 32

 @[x;,1;*;100]
0 1
1000 1100 1200
```

```
20
30 31 32

 @[x;1 2;*;100]
0 1
1000 1100 1200
2000
30 31 32
```

10.3 dmend ⇒ .[x;i;f[;y]]

.[x;i;f] examples

```
 x:(`x00`x01;`x10`x11`x12;`x20;`x30`x31`x32);x
x00 x01
x10 x11 x12
x20
x30 x31 x32

 .[x;1 2;`newValue]
x00 x01
x10 x11 newValue
x20
x30 x31 x32
```

.[x;i;f;y] examples

```
 x:(`x00`x01;`x10`x11`x12;`x20;`x30`x31`x32);x
x00 x01
x10 x11 x12
x20
x30 x31 x32

 i:(1 3; 0 1);i
1 3
0 1

 y:(`a`b;`c`d);y
a b
c d

 .[x;i;:;y]
x00 x01
a b x12
```

```
x20
c d x32

 x:(0 1;10 11 12;20;30 31 32);x
0 1
10 11 12
20
30 31 32

 .[x;i;*;-1]
0 1
-10 -11 12
20
-30 -31 32
```

11 I/O

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.

11.1 Input format values to table

This section shows you the syntax for reading in data into a table with the correct type.

```
 d:,(`date`time`int`float`char`symbol)            / headers
 d,:,(2020.04.20;12:34:56.789;37;12.3;"hi";`bye)) /data
 d
date       time         int float char  symbol
2020-04-20 12:34:56.789 37  12.3  hi    bye

 `csv'd                                           / to csv
date,time,int,float,char,symbol
2020-04-20,12:34:56.789,37,12.3,"hi",bye

 "some.csv"0:`csv'd                               / write to some.csv
 0:"some.csv"                                     / read from some.csv
date,time,int,float,char,symbol
2020-04-20,12:34:56.789,37,12.3,"hi",bye

 ("Dtifs*";,",")0:"some.csv"                      / read into table
date       time         int float char    symbol
---------- ------------ --- ----- ------- ------
2020-04-20 12:34:56.789  37 12.3  "hi"    bye
```

11.2 Format to CSV/json/k ⇒ 'csv x

Convert x to CSV/json/k format. Works on atoms, lists, and tables.

```
 `csv 3 1 2
"3,1,2"
 `json (3;`abc;2.3;"blue")
"[\"3\",\"abc\",2.3,\"blue\"]"
 `k [[]i:!5;s:`a`b`c`d`e;v:5?10]
"[[]i:0 1 2 3 4;s:`a`b`c`d`e;v:7 4 7 3 2]"
 `csv `a`b!((1 2);(3 4))      / error as dictionary input
:class
```

11.3 write line ⇒ 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
 ""0:$'("blue";"red";3)  / each element to string
blue
red
3
 "some.csv"0:,`csv 3 1 2 / will fail without enlist
```

11.4 read line ⇒ 0:x

Read from file x.

```
 "some.txt"0:,`csv 3 1 2 / first write a file to some.txt
 0:"some.txt"            / now read it back
3,1,2
```

11.5 write char ⇒ x 1:y

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

```
 "some.txt"1:"hello here\nis some text\n"
 1:"some.txt"
"hello here\nis some text\n"
  t:+`a`b!(1 2;3 4);t
a b
- -
1 3
2 4
 "some.k"1:`k t      / write table to file in k format
```

11.6 read char ⇒ 1:x

Read from file x.

```
 "some.txt"0:,`csv 3 1 2 / first write a file to some.txt
 1:"some.txt"            / now read it back
"3,1,2\n"
  t:+`a`b!(1 2;3 4);t
a b
- -
1 3
2 4
 t:`k?1:"some.k";t   / read file stored in k format (as shown above)
a b
- -
1 3
2 4
```

11.7 Load file ⇒ 2: x

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
0.3329232  0.3528549  0.2659892  0.1927104  0.4304933
0.05392223 0.7969098  0.4312251  0.7799843  0.5060091
0.1922035  0.5056334  0.2600317  0.4555804  0.56671
0.4545242  0.01599503 0.1710724  0.4320832  0.4520696
0.7843445  0.4319026  0.1015124  0.877304   0.9949587
0.09920892 0.8340988  0.3119439  0.4945446  0.967994
0.5899243  0.4547598  0.436347   0.3572658  0.2969937
0.3565662  0.7649578  0.5738509  0.402629   0.7585447
0.4563912  0.509789   0.01807586 0.3083831  0.2447315
0.1614906  0.601976   0.1165871  0.0395344  0.05975276
0.1710438  0.1687449  0.7200667  0.9578548  0.7333167
0.8933161  0.7996999  0.1117325  0.2385556  0.5339807
0.03895895 0.4215705  0.01501522 0.9872831  0.9973345
0.4643205  0.5794769  0.5476008  0.8957309  0.1633682
0.1797837  0.5683136  0.993727   0.1164099  0.3229972
0.3687319  0.8430398  0.5818712  0.5021431  0.8034257
0.9274384  0.6739888  0.1821047  0.113806   0.9466886
0.8766261  0.05144491 0.8987524  0.2241464  0.617475
0.455943   0.449666   0.9678184  0.06839654 0.1232913
..
```

11.8 Save/load ⇒ x 2: y

The function is used to save data and load shared libraries. (1) Save to file x non-atomic data y (eg. lists, dictionaries, or tables). (2) Load shared library x with dictionary y.

(1) Save to file. 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 (1s) and read (13ms) 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
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
0.3329232  0.3528549  0.2659892  0.1927104  0.4304933
0.05392223 0.7969098  0.4312251  0.7799843  0.5060091
0.1922035  0.5056334  0.2600317  0.4555804  0.56671
0.4545242  0.01599503 0.1710724  0.4320832  0.4520696
0.7843445  0.4319026  0.1015124  0.877304   0.9949587
0.09920892 0.8340988  0.3119439  0.4945446  0.967994
0.5899243  0.4547598  0.436347   0.3572658  0.2969937
0.3565662  0.7649578  0.5738509  0.402629   0.7585447
0.4563912  0.509789   0.01807586 0.3083831  0.2447315
0.1614906  0.601976   0.1165871  0.0395344  0.05975276
0.1710438  0.1687449  0.7200667  0.9578548  0.7333167
0.8933161  0.7996999  0.1117325  0.2385556  0.5339807
0.03895895 0.4215705  0.01501522 0.9872831  0.9973345
0.4643205  0.5794769  0.5476008  0.8957309  0.1633682
0.1797837  0.5683136  0.993727   0.1164099  0.3229972
0.3687319  0.8430398  0.5818712  0.5021431  0.8034257
0.9274384  0.6739888  0.1821047  0.113806   0.9466886
0.8766261  0.05144491 0.8987524  0.2241464  0.617475
0.455943   0.449666   0.9678184  0.06839654 0.1232913
..

 ,"r"
 \\
bash-3.2$ ./k
Sep 13 2020 16GB (c) shakti
 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
0.3329232  0.3528549  0.2659892  0.1927104  0.4304933
0.05392223 0.7969098  0.4312251  0.7799843  0.5060091
0.1922035  0.5056334  0.2600317  0.4555804  0.56671
0.4545242  0.01599503 0.1710724  0.4320832  0.4520696
0.7843445  0.4319026  0.1015124  0.877304   0.9949587
0.09920892 0.8340988  0.3119439  0.4945446  0.967994
0.5899243  0.4547598  0.436347   0.3572658  0.2969937
0.3565662  0.7649578  0.5738509  0.402629   0.7585447
0.4563912  0.509789   0.01807586 0.3083831  0.2447315
0.1614906  0.601976   0.1165871  0.0395344  0.05975276
```

```
0.1710438  0.1687449  0.7200667  0.9578548  0.7333167
0.8933161  0.7996999  0.1117325  0.2385556  0.5339807
0.03895895 0.4215705  0.01501522 0.9872831  0.9973345
0.4643205  0.5794769  0.5476008  0.8957309  0.1633682
0.1797837  0.5683136  0.993727   0.1164099  0.3229972
0.3687319  0.8430398  0.5818712  0.5021431  0.8034257
0.9274384  0.6739888  0.1821047  0.113806   0.9466886
0.8766261  0.05144491 0.8987524  0.2241464  0.617475
0.455943   0.449666   0.9678184  0.06839654 0.1232913
..
```

(2) 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"2:{add1:"i";add2:"i";indx:"Ii"}
 f[`add1] 12
13
 f[`indx][12 13 14;2]
14
```

11.9 conn/set ⇒ 3:

TBD

11.10 http/get ⇒ 4:

TBD

12 Tables

This chapter introduces the different types of tables available in k9. Table, XTab and TTab are very similar and as you'll see in the kSQL chapter are easy to query. In the benchmark chapter tables were shown to be fast to save, read, and query.

[Table], page 57, [[]a:..;..]
[XTab], page 58, `a..![[]..]
⟨undefined⟩ [TTab], page ⟨undefined⟩, [[a:..]..]

12.1 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.

12.1.1 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 defintion with square brackeets 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 TTab. 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 values but be a list
[[] a:1; col2:3.6]

:class

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

12.1.2 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
```

12.1.3 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
```

12.2 XTab

An cross tab (XTab) 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
```

12.3 TTab

TTab 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

13 kSQL

kSQL is a powerful query language for tables. The benchmark chapter has shown how quickly k9 can process big tables. Here we'll work on a small table in order to make things clear and repeatable.

Here's a small sample table.

```
 t:[[]day: 2020.01.06 2020.01.06 2020.01.07 2020.01.07;sym:`a`b`a`b;price:100 50 101 4
day        sym price
---------- --- -----
2020.01.06 a     100
2020.01.06 b      50
2020.01.07 a     101
2020.01.07 b      49
```

kSQL looks similar to SQL as can be shown from the built-in reference screen.

`select A by B from T where C`

```
 select from t
day        sym price
---------- --- -----
2020.01.06 a     100
2020.01.06 b      50
2020.01.07 a     101
2020.01.07 b      49

 select from t where sym=`a
day        sym price
---------- --- -----
2020.01.06 a     100
2020.01.07 a     101

 select avg price by sym from t
sym|price
---|-----
a  |100.5
b  |49.5
```

13.1 Joins

k9 has a number of methods to join tables together which are described below. In this section T1 and T2 represent tables and TT1 and TT2 represent TTabs.

join	**x**	**y**
[union], page 60,	table	table
[left], page 61,	table	TTab
[outer], page 62,	TTab	TTab
[asof], page 62,	table	TTab (by time)

13.1.1 union join ⇒ T_1,T_2

Union join table T1 with table T2.

```
 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
```

13.1.2 left join ⇒ T,TT

Left join table T with TTab TT. Result includes all rows from T and values from T where there is no TT value.

```
 T:[[]s:`a`b`c;p:1 2 3;q:7 8 9]
 TT:[[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

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

 T,TT
```

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

13.1.3 outer join ⇒ TT_1,TT_2

Outer join TTab TT1 with key TTab TT2.

```
 TT1:[[s:`a`b]p:1 2;q:3 4]
 TT2:[[s:`b`c]p:9 8;q:7 6]
 TT1
s|p q
-|- -
a|1 3
b|2 4

 TT2
s|p q
-|- -
b|9 7
c|8 6

 TT1,TT2
s|p q
-|- -
a|1 3
b|9 7
c|8 6
```

13.1.4 asof join ⇒ T,TT

Asof joins a table T to a TTab TT (key by time) such that the T values show the preceeding or equal time value of TT.

```
 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

 TT:[[t:09:32 09:41 09:45]q:50 51 52];TT
t    |q
-----|--
09:32|50
09:41|51
```

```
09:45|52

 T,TT
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;TT:[[t:n?`t 0]p:n?100];5#TT
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,TT
222
```

13.2 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.

13.2.1 insert ⇒ 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
```

13.2.2 upsert ⇒ TT,d

Insert dictionary d into TTab TT.

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

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

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

14 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.

14.1 load ⇒ \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
```

14.2 timing ⇒ \t

List time elapsed

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

14.3 variables ⇒ \v

List variables

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

14.4 memory ⇒ \w

List memory usage

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

14.5 cd ⇒ \cd x

Change directory (cd) into x

```
 \cd scripts
```

15 Errors

Given the terse syntax of k9 it likley 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 66, [rank], page 66, [length], page 66, [_type], page 66, [domain]
page 66, [limit], page 66, stack ([parse], page 67, [_value], page 67)
```

15.1 :class

Calling a function on mismatched types.

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

15.2 rank

Calling a function with too many parameters.

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

15.3 length

Operations on unequal length lists that require equal length.

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

15.4 :type

Calling a function with an unsupported variable type.

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

15.5 :domain

Exhausted the number of input values

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

15.6 :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
```

15.7 :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
```

15.8 :parse

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

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

15.9 :value

Undefined variable is used.

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

16 Conculsion

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.

Table of Contents