k9 simples

John Estrada

This manual is for Shakti (k9) build 2020.04.14.
25 April 2020

Copyright © 2020 John Estrada

Table of Contents

k9: Manual

This document explains the usage of the k9 programming language in very simple terms and is intended for newer developers. k9 is a rapidly evolving platform therefore newer versions may have additional functionality not covered in this document.

1 Intro

Shakti, aka k9, is a programming language built for speed, consice syntax, and data manipulation. The syntax is a bit special and although it might feel like an impediment at first becomes an advantage with use.

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 and similar languages, then you should likely appreciate symbols is shorter but looks like line noise. That's true but so did arithetic until you learns 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.

1.1 Get k9.

https://shakti.com/

Go to the Shakti website and click on download. You'll need to enter a few pieces of information and then you'll have a choice to download either a Linux or MacOS version. Click on the required OS version and you'll download a `k.zip` file around 50 kb in size. Unzip that file and you'll have a single executable file `k` which is the language.

1.2 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
2020.04.01 (c) shakti
 //

 ./k
2020.04.01 (c) shakti
 //
```

1.3 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
2020.04.01 (c) shakti
 n:10000                      / n data points
 s:`a`b`c                     / data for symbols a, b, and c
 q:+s!(-1+n?2;-1+n?2;-1+n?2) / 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                       / calulate the return of each symbol
a   b   c
--- --- --
-68 117 73
 {(+/m*m:x-avg x)%#x}'+q      / calulate the variance of each symbol
a|0.6601538
b|0.6629631
c|0.6708467
```

1.4 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
```

1.5 k9 nuances

One will need to understand some basic rules of k9 in order to progress. These will likely seem strange at first.

1.5.1 : 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.

1.5.2 % is used to divide numbers

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

1.5.3 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.

1.5.4 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.

1.5.5 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
```

1.5.6 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.

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

1.5.7 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)`

1.6 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.

```
 \
$k f.k

Verb                    Adverb                Atom            Type  System
:  set                  '  each       i bar   bool  110b         b  \k
+  plus       flip      /  over/right i div   int   2 3 4        i *\l a.k
-  minus      negate    \  scan/left  i mod   float 2e3 0n 0w    f  \v [d]
*  times      first     ': eachprior          date  2024.01.01   D *\f [d]
%  divide               /: [n]over    i sv    time  12:34:56.789 t  \w [x]
&  min        where     \: [n]scan    i vs    char  "ab "        c  \t:n x
|  max        reverse                         sym   `a`b`        s  \u:n x
<  less       asc        I/O
>  more       dsc       0: readwrite  line    List (2;3.4;`c)    L  \fl line
=  equal      group     1: readwrite  char    Dict [a:2;b:`c]   ??  \fc char
~  match      not       2: write      data    Func {(+/x)%#x}    .
!  key        enum      3: *conn/set
,  cat        enlist    4: *http/get
^  cut        sort
$  cast       string    $[c;t;f]      cond
#  take       count     #[t;c;b[;a]]  select   table [[]a:`b`c]   T
_  drop       floor    *_[t;c;b[;a]]  update  Ttable [[a:..]b:]  TT
?  find       unique   *?[x;i;f[;y]]  splice  Stable S! [[]...]  ST
@  at         type      @[x;i;f[;y]]  amend                          \cd dir
.  apply      value     .[x;i;f[;y]]  dmend                          \\  exit

A_-~%$ L|+#*&=<>^?! ,@.
A+-*%&|<>=$ L,#_^?! ~@.  F#_

count first last min max sum avg; in bin within; *exp log sin cos
select A by B from T where C; delete from T where C

/comment \display [dict] :expr (leading space)
roundtrip: `json?`json(2.3;.z.D;.z.t;"abc")  / also: `csv`k
time/cuanto: 2m 2d ..   12:34:56.123456789 e.g. .z.D+2m / .z.[tuv]
date/cuando: 2024.01.01T12:34:56.123456789 e.g.  7\.z.D / .z.[TUV]

error: parse value class rank type domain length limit
limit: sym8(*256) {[param8]local8 global32 const128 jump256}
```

2 Data / Nouns

The basic data types of the k9 language are numbers (interger 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.

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`s`t`D
```

2.1 Numeric Data

Numbers can be stored as integers and floats.

```
 @3
`i
 @3.1
`f
 a:3;b:3.1;
 @a
`i
 @b
`f
```

Numeric data can be recast as other types using the `$` command. The `$` is an overloaded function and can either convert to string or cast to type if provided. The first usage take only a single argurment, the item to be converted to a string, while the second usage takes two arguments, the items to be converted and also the type to convert into.

You'll note that the string of 3 is represented as ,"3". The comma represents the string is a list of length one and not a single element.

```
 $3     / convert to string
,"3"
 $12    / convert to string
"12"
 `f$12  / convert to float
12f
 `t$12  / convert to time
```

```
00:00:00.012
 `d$12  / convert to date
2024-01-13
```

2.2 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.

```
 @"b"
`c
 @"bd"
`C
 @`blue
`s
 @`blue`red
`S
```

2.3 Temporal Data

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

```
 @12:34:56.789             / time
`t
 @2020.04.20               / date
`D
 @2020.04.20T12:34:56.789  / date and 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$00:00:00.001           / numeric representation of 1s
1000
 `i$00:01:00.000           / numeric representation of 1m
60000
 `t$12345                  / convert milliseconds to time
00:00:12.345
 .z.D                      / current date in GMT
2020-04-17
 `i$.z.D                   / numeric representation of date
-1351
 `i$2024.01.01             / zero date
0
 `D$0                      / zero date
```

```
2024.01.01
```

2.4 Extreme values

Data types can not only represent in-range values but also null and out-of-range values.

type	**null**	**out of range**
i	0N	0W
f	0n	0w

3 Functions / Verbs

This chapter explains all functions, aka verbs. Most functions are overloaded and change depending on the number and type of arguments.

3.1 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
```

3.2 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
error: 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
```

3.3 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
```

3.4 minus ⇒ x-y

Subtract y from x.

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

3.5 negate ⇒ -x

Negative 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
```

3.6 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
```

3.7 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
```

3.8 divide ⇒ x%y

Divide x by y.

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

3.9 min ⇒ 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
```

3.10 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
```

3.11 max ⇒ 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
```

3.12 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
```

3.13 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
```

3.14 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
```

3.15 equal ⇒ x=y

x equal to y

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

3.16 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
```

3.17 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
```

3.18 not ⇒ ~x

Boolean invert of x

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

3.19 key ⇒ 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
```

3.20 enum ⇒ !x

Generate an integer list from 0 to x-1.

```
 !3
0 1 2
```

3.21 cat ⇒ x,y

Concatenate x and y.

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

3.22 enlist ⇒ ,x

Create a list from x

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

3.23 cut ⇒ x^y

Reshape a list y by indices x.

```
 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
```

3.24 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
```

3.25 cast ⇒ x$y

Cast y into type x.

```
 `i$37.1 37.9
37 37
 `f$3
3f
 `D$"2020.03.01"
2020-03-01
 `t$123
00:00:00.123
```

3.26 string ⇒ $x

Cast x to string.

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

3.27 take ⇒ x#y

First (last) x elements of y if x is positive (negative). If x is a list then returns any values in both x and y.

```
 3#0 1 2 3 4 5          / first three
0 1 2
 -3#0 1 2 3 4 5         / last three
3 4 5
 2#"hello"
"he"
 (1 2 3 7 8 9)#(2 8 20) / union
2 8
```

3.28 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
```

3.29 drop ⇒ x_y

Return the list y without the first (last) x elements if x is positive (negative). If x is a list then returns any values from y not in x.

```
 3_0 1 2 3 4 5
3 4 5
 -3_0 1 2 3 4 5
0 1 2
 a:3;b:0 9 1 8 2 7;
 a_b
8 2 7
 (1 2 3 7 8 9)_(2 8 20)
,20
```

3.30 floor ⇒ _x

Return the integer floor of float x.

```
 _3.7
3
```

3.31 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
```

3.32 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"
```

3.33 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
```

3.34 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
```

3.35 apply ⇒ x.y

Given list x return the value at list y. The action of apply 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
```

3.36 value ⇒ .x

- x= dictionary ⇒ Return the value of x as lists.
- x= parse output ⇒ Evaulate x.

```
  `a`b!(1 2;3 4)
a|1 2
b|3 4
 .`a`b!(1 2;3 4)
a   b
1 2 3 4
 p::3+2
 p
+
3
2
 . p   / the space between . and p is necessary
5
```

4 Function Modifiers / Adverbs

k9 uses function modifiers / adverbs in order to have functions operate iteratively over lists.

```
Adverb
'  See [each], page 21,           See [bar], page 21.
/  See [over], page 22,See [right], page 22, See [div], page 22.
\  See [scan], page 21,See [left], page 21,  See [mod], page 22.
': See [eachprior], page 23.
/: See [[n]over], page 23.             See [sv], page 23.
\: [n]scan             See [vs], page 23.
```

4.1 each ⇒ f'x

Apply each value in list x to function f.

```
 *((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
```

4.2 bar ⇒ x'y

y divided by x using integer division and then multiplied by x. x is an integer and y is a list of integers.

```
 5'0 1 2 3 4 5 9 10
0 0 0 0 0 5 5 10
```

4.3 scan ⇒ (f\)x

Create values 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]

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

4.4 left ⇒ f\[x;y]

Apply f[;y] to each value in x.

```
 {x+y}[100 200 300;1 2 3]  / add the lists together itemize
101 202 303
 {x+y}\[100 200 300;1 2 3] / add the list y to each value of x
101 102 103
201 202 203
301 302 303
 {x,y}\[`l1`l2`l3;`r1`r2`r3]
l1 r1 r2 r3
l2 r1 r2 r3
l3 r1 r2 r3
```

4.5 mod ⇒ x\y

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

```
 12\27
3
 5\22
2
```

4.6 over ⇒ (f/)x

Same as scan but only print last value.

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

4.7 right ⇒ f/[x;y]

Apply f[x;] to each value in y.

```
 {x+y}[100 200 300;1 2 3]  / add the lists together itemize
101 202 303
 {x+y}/[100 200 300;1 2 3] / add the list y to each value of x
101 201 301
102 202 302
103 203 303
 {x,y}/[`l1`l2`l3;`r1`r2`r3]
l1 l2 l3 r1
l1 l2 l3 r2
l1 l2 l3 r3
```

4.8 div ⇒ x/y

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

```
 2/7
3
 5/22
4
```

4.9 eachprior ⇒ f':[x;y]

Apply f[y_n;y_{n-1}]. f_0 is a special case of 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
```

4.10 [n]over ⇒ 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]]

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

4.11 vs ⇒ x\:y

Convert y (base 10) into base x.

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

4.12 sv ⇒ x/:y

Convert list y (base x) into base 10.

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

5 Lists

k9 is optimized for operations on uniform lists of data. In order to take full advantage one should store data in lists and operate on them without iteration.

5.1 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
error: type
```

5.2 List Indicing

Lists can be indexed by using a few notations.

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

5.3 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;identify 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
error: length
 @[!10;1 3;:;10 20 30]  / index and value array length mistmatch
error: length
```

6 Dictionaries and Dictionary Functions

Dictionaries are key-value pairs of data. The values in the dictionary can be single elements or lists.

6.1 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
 0|37.4
10|46.3
 1|0.1
```

6.2 Dictionary Indicing ⇒ x@y

Dictionary indicing, 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 indicing work, output surpressed
 x@`b;  / at
 x(`b); / parenthesis
 x `b;  / space
 x[`b]; / square bracket
```

6.3 Dictionary Key ⇒ !x

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

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

6.4 Dictionary as Value ⇒ .x

A dictionary can be returned as values using the . function. The function returns a list of length two. The first element is a list of the keys. The second element is a list of the values.

```
 . 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 indicing 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)
time              temp
12:00 12:01 12:10 25 25.1 25.6

 (. d1)[1]
12:00 12:01 12:10
25    25.1  25.6

 (. d1)[1][1]
25 25.1 25.6
 (. d1)[1][1][0]
25f
```

6.5 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
```

6.6 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
```

6.7 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
```

6.8 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
```

6.9 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
```

7 More functions

This chapter includes functions that likely will be included elsewhere later.

7.1 in ⇒ x'y

Determine if y is in list x.

```
 `a`b`d`e'`c
0b
 `a`b`d`e'`b
1b
 (!10)'2
1b
 (!10)'2. / will error as type is different
(!10)'2.
^
error: type
```

7.2 parse ⇒ :x

Parse allows one to see how a command is parsed into normal k9 form. One can value the parse by using the value command, See [value], page 20.

```
 :3+2
+
3
2
 t:+`a`b!(1 2;3 4)
 :select from t
t
 :select a from t
t
[..]
 p::select a from t  / store output into p
 #p
2
 p 0
`t
 p 1
a|a
 . p                 / value parse expression
a
-
1
2
 .(`t;`a!`a)         / value expression
a
-
```

```
1
2
 select from t       / orginal statement
a
-
1
2
```

Now for an example with a group clause.

```
 t:+`a`b`c!(`x`y`x;0 2 10;1 1 0)
 select avg:+/b%#b by a from t
a|avg
-|---
x|5
y|2
 p::select avg:+/b%#b by a from t
 .(#;`t;();`a!`a;`avg!(%;(+/;`b);#`b)) / parse form
 #[t;();`a!`a;`avg!(%;(+/;`b);#`b)]    / functional form
a|avg
-|---
x|10
y|2
 p::select avg:+/b%#b by a from t where c=1
 #[t;(=;`c;1);`a!`a;`avg!(%;(+/;`b);#`b)]
a|avg
-|---
x|0
y|2
```

In the example above the parse output is reduced. In order to see the elements in the output one could manually return the values in the list, eg. p[2;0 1 2].

7.3 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
```

7.4 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
```

7.5 Histogram ⇒ ‘freq

Compute a histogram of a list.

```
 ``freq x:100000?10
0| 9907
1| 9963
2| 9938
3|10063
4|10018
5|10007
6|10037
```

```
7|10036
8| 9907
9|10124
 ^#'=x   / same result but slower
0| 9907
1| 9963
2| 9938
3|10063
4|10018
5|10007
6|10037
7|10036
8| 9907
9|10124
```

8 I/O

Functions for input and ouput (I/O).

8.1 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
```

8.2 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
error: class
```

8.3 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
```

8.4 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
```

8.5 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
```

8.6 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
```

8.7 write data ⇒ 2:

TBD

8.8 conn/set ⇒ 3:

TBD

8.9 http/get ⇒ 4:

TBD

9 Tables and kSQL

This chapter introduces k9 tables and the kSQL language to query.

9.1 Tables

Here is an example of a table with three columns (Day, Weather, and Temp) and three rows.

```
 t:[[]Day:2020.04.10+!3;Weather:`sunny`cold`sunny;Temp:22 12 18]
 t
Day        Weather Temp
---------- ------- ----
2020-04-10 sunny     22
2020-04-11 cold      12
2020-04-12 sunny     18
 @t                              / tables are type `A (`t is for time)
`A
 +t
Day    |2020-04-10 2020-04-11 2020-04-12
Weather|sunny cold sunny
Temp   |22 12 18
```

9.2 A_Tables

Here is an example of a A_table with three columns (Day, Weather, and Temp) and three rows. One column (Day) will be add as a key.

```
 t:[[Day:2020.04.10+!3]Weather:`sunny`cold`sunny;Temp:22 12 18]
 t
Day       |Weather Temp
----------|------- ----
2020-04-10|sunny     22
2020-04-11|cold      12
2020-04-12|sunny     18

 @t                         / A_tables have type `AA
`AA
```

9.3 S_Tables

TBD

```
 x:`a`b! [[]c:2 3;d:3 4;e:4 5]
 x
 |c d e
-|- - -
a|2 3 4
b|3 4 5
```

```
 @x                                / S_tables are type `SA
`SA
```

9.4 kSQL

kSQL is a powerful query language for tables.

```
 select |/Temp from t where Weather=`sunny
Temp|22

 select from t where Weather=`sunny
Day        Weather Temp
---------- ------- ----
2020-04-10 sunny     22
2020-04-12 sunny     18

 select {+/x%#x}Temp from t where Weather=`sunny
Temp|20

 select |/Temp from t where Weather=`sunny
Temp|22
```

9.5 Joins

Joining tables together. In this section x, y represent tables and kx and ky represent keyed/A_tables.

join	**x**	**y**
union	table	table
left	table	Atable
outer	Atable	Atable
asof	table	Atable (by time)

9.5.1 union join ⇒ x,y

Union join table x with table y.

```
 x:[[]s:`a`b;p:1 2;q:3 4]
 y:[[]s:`b`c;p:11 12;q:21 22]
 x
s p q
- - -
a 1 3
b 2 4

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

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

9.5.2 left join ⇒ x,y

Left join table x with keyed table/A_table. Result includes all rows from x and values from x where there is no y value.

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

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

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

9.5.3 outer join ⇒ x,y

Outer join key table/A_table x with key table/A_table y.

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

```
 y
s|p q
-|- -
b|9 7
c|8 6

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

10 System

This chapter describes the system settings and functions.

10.1 Display ⇒ \k

```
 m:(100 101 102;3;14 15);d:`a`b!(1 2;3 4)
 \k 0
 m
100 101 102
3
14 15

 d
a|1 2
b|3 4

 \k 1
 m
(100 101 102;3;14 15)
 d
[a:1 2;b:3 4]
```

10.2 Variables ⇒ \v

List variables

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

10.3 Memory ⇒ \w

List memory usage

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

10.4 Timing ⇒ \t

List time elapsed

```
 \t ^(`i$1e7)?`i$1e8
360
```

11 Errors

This section contains information on the various error messages in k9.

11.1 error: parse

Syntax is wrong.

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

11.2 error: value

Undefined variable is used.

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

11.3 error: class

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

11.4 error: length

Operations on unequal length lists that require equal length.

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

11.5 error: domain

Exhausted the number of input values

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