k9 simples

John Estrada

This manual is for Shakti (k9) build 2020.05.08.
16 May 2020

Copyright © 2020 John Estrada

Table of Contents

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 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. *\l a.k.

```
 \
Verb                    Adverb                Noun            Type  System
:                       '  each               bool  011b        b  \k
+  plus       flip      /  over/right         int   2 3 4       i  \l a.k
-  minus      minus     \  scan/left          float 2.3 0w      f  \v [d]
*  times      first     ': eachprior          char  " ab"       c *\f [d]
%  divide               /: [n]over            sym   ``a`b       s  \w [x]
&  and        where     \: [n]scan            time  12:34:56.789 t  \t:n x
|  or         reverse                         date  2024.01.01  D  \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: readwrite  data    Func {(+/x)%#x}   .
!  key        key       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       uniq      *?[x;i;f[;y]] splice  Stable S! [[]...]  ST
@  at         type      @[x;i;f[;y]]  amend                          \cd dir
.  dot        dot       .[x;i;f[;y]]  dmend                          \\  exit

       Atom(pervasive)   List/Dict     Func     ALDF  tolerant    random
monad: $-~_              .!#*|+&^<>=?           @,     [*|&!]a     ?a
dyad:  $+-*%&|<>= L[@?]  .!#_^,        f[#_]    @~    a[,#_!]a    a?x

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

roundtrip: `json?`json@(2.3;"abc";.z.T) `k?`k@x `csv?`csv@x
time/cuanto: 2m 2d ..   12:34:56.123456789 .z.[dtv]
date/cuando: 2024.01.01T12:34:56.123456789 .z.[DTV]

error: nyi class rank length type domain limit (value stack parse value)
limit: sym8(*256) 4K {[param8]local8 global32 const128 jump256}
atom/list/dict/table (scalar/vector/matrix/tensor) dyad/monad noun/verb/adverb
/comment \display [dict] :expr (leading space) OVERLOAD(?)
```

1.3 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
 //
```

1.4 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
```

1.5 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
```

1.6 k9 nuances

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

1.6.1 The language is being changed 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.

1.6.2 : 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.

1.6.3 % is used to divide numbers

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

1.6.4 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.

1.6.5 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.

1.6.6 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
```

1.6.7 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 lits or function is via the type function. Lists and functions do not have the same type.

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

1.6.8 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

Examples of k9 in practice.

2.1 Data Manipulation

Generate a table of financial random data and compute basic statistics quickly.

```
n:_50*1000*1000                          / 50 million rows
t:{g@<g:09:00:00.000+x?10:00:00.000}     / random times
s:{x?`a`b`c`d`e}                         / random symbols
d:{(-6+!13)@(x?_2e9)bin 1e9*(0,(1-|d),1+d:0.682689492 0.954499736 0.997300203 0.99993
q:+`t`s`d!(t[n];s[n];d[n])               / generate the data
```

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

```
 select min t, max t from q
t|09:00:00.002
t|18:59:59.998

 select #s by s from q
s|s
-|--------
a|10002646
b|10001228
c|10002255
d| 9993125
e|10000746

 select #d by d from q
d |d
--|--------
-5|       9
-4|    1572
-3|   65761
-2| 1069559
-1| 6794024
 0|34133178
 1| 6799826
 2| 1068860
 3|   65582
 4|    1613
 5|      16
```

2.2 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)
```

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)    / compute and 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";"++"]
^
error: 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 26 27 28 29 30 31
```

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 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.

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

```
    Atom             Type
See [bool], page 10,  110b         b
See [int], page 10,   2 3 4        i
See [float], page 11, 2e3 0n 0w    f
See [date], page 11,  2024.01.01   D
See [time], page 11,  12:34:56.789 t
See [char], page 12,  "ab "        c
See [sym], page 12,   `a`b`        s
```

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
```

3.1 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
```

3.2 Numeric Data

Numbers can be stored as integers and floats.

3.2.1 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
```

3.2.2 float ⇒ Float f

Float

```
 3.1
3.1
 3.1+1.2
4.3
 3.1-1.1   / looks like an int but really is a float
2
 @3.1-1.1
1f
 @3.1
`f
 a:3.1;
 @a
`f
```

3.3 Temporal Data

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

3.3.1 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
2020-04-17
 `i$.z.D                    / numeric representation of date
-1351
 `i$2024.01.01              / zero date
0
 `D$0                       / zero date
2024.01.01
```

3.3.2 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$00:00:00.001            / 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
```

3.3.3 datetime ⇒ Datetime T

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

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

3.4 Text Data

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

3.4.1 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
```

3.4.2 sym ⇒ Symbol s

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

```
 @`blue
`s
 @`blue`red
`S
```

3.5 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

4 Functions / Verbs

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

4.1 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
```

4.2 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
```

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 minus ⇒ x-y, -x

Subtract y from x.

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

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
```

4.5 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
```

4.6 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
```

4.7 divide ⇒ x%y

Divide x by y.

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

4.8 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
```

4.9 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
```

4.10 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
```

4.11 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
```

4.12 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
```

4.13 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
```

4.14 equal ⇒ x=y

x equal to y

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

4.15 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
```

4.16 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
```

4.17 not ⇒ ~x

Boolean invert of x

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

4.18 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
```

4.19 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 0 0 0  0  0  0  0  0  0 1 1 1 1 1 1 1 1 1 1  1  1  1  1  1  1 2 2
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 13 14 15 0 1
 18#+!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
0 0 12
0 0 13
0 0 14
0 0 15
0 1  0
0 1  1
 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
```

4.20 cat ⇒ x,y

Concatenate x and y.

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

4.21 enlist ⇒ ,x

Create a list from x

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

```
 3~,3
0b
```

4.22 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
```

4.23 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
```

4.24 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
```

4.25 string ⇒ $x

Cast x to string.

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

4.26 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
```

4.27 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
```

4.28 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
```

4.29 floor ⇒ _x

Return the integer floor of float x.

```
 _3.7
3
```

4.30 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
```

4.31 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"
```

4.32 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
```

4.33 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
```

4.34 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
```

4.35 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
```

5 Function Modifiers / Adverbs

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

```
Adverb
'  See [each], page 26.
/  See [over], page 27,See [right], page 27.
\  See [scan], page 26,See [left], page 26.
': See [eachprior], page 27.
/: See [c over], page 28,See [n over], page 28.
\: See [c scan], page 28,See [n scan], page 27.
```

5.1 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
```

5.2 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
```

5.3 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
```

5.4 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
```

5.5 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
```

5.6 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
```

5.7 n scan ⇒ 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 collapse
 `x f\: `y0`y1`y2
x     y0
xy0   y1
xy0y1 y2
```

5.8 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 0.8514578 0.7249803 0.5255965 0.2762517 0.07631498 0.00
```

5.9 n over ⇒ x f/:y

Same as n scan but only return last value.

5.10 c(onverge) over ⇒ f/:x

Same as converge scan but only return last value.

5.11 vs ⇒ x\:y

Convert y (base 10) into base x.

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

5.12 sv ⇒ x/:y

Convert list y (base x) into base 10.

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

6 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.

6.1 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
```

6.2 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
```

6.3 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
```

7 Dictionaries and Dictionary Functions

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

7.1 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
```

7.2 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
```

7.3 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
```

7.4 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 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[]
12:00 12:01 12:10
25    25.1  25.6

 d1[][1]
25 25.1 25.6
 d1[][1;0]
25f
```

7.5 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
```

7.6 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
```

7.7 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
```

7.8 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
```

7.9 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
```

8 More functions

Functions not listed as verbs.

8.1 Math Functions ⇒ sqrt exp log sin cos

Log is natural log and sin and cos are in radians.

```
 sqrt 2
1.414214
 exp 1
2.718282
 log 10
2.302585
 sin 0
0f
 cos 0
1f
```

8.2 Various Functions ⇒ ema div mod bar in bin within

8.2.1 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
```

8.2.2 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
```

8.2.3 bar ⇒ x bar 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
```

8.2.4 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
```

8.2.5 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
```

8.2.6 within ⇒ x within y

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

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

8.3 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
```

8.4 cond ⇒ $[x;y;z]

If x then y else z.

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

8.5 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 24.

```
 :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].

8.6 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
```

8.7 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
```

9 I/O

Functions for input and ouput (I/O).

9.1 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
```

9.2 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
```

9.3 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
```

9.4 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
```

9.5 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
```

9.6 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
```

9.7 write data ⇒ 2:

TBD

9.8 conn/set ⇒ 3:

TBD

9.9 http/get ⇒ 4:

TBD

10 Tables and kSQL

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

10.1 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
```

10.2 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
```

10.3 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
```

10.4 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
```

10.5 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)

10.5.1 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
```

10.5.2 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
```

10.5.3 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
```

11 System

This chapter describes the system settings and functions.

11.1 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]
```

11.2 Load File⇒ \l

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
error: nyi
```

11.3 Variables ⇒ \v

List variables

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

11.4 Memory ⇒ \w

List memory usage

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

11.5 Timing ⇒ \t

List time elapsed

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

12 Errors

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

12.1 error: class

Calling a function on mismatched types.

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

12.2 error: domain

Exhausted the number of input values

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

12.3 error: length

Operations on unequal length lists that require equal length.

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

12.4 error: nyi

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

2020.04.08 (c) shakti

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

2020.05.03 (c) shakti

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

12.5 error: parse

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

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

12.6 error: rank

Calling a function with too many parameters.

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

12.7 error: type

Calling a function with an unsupported variable type.

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

12.8 error: value

Undefined variable is used.

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