The KeRF Programming Language
Work In Progress

John Earnest

November 19, 2015

This manual is a reference guide to KeRF, a concise multi-paradigm language with an emphasis on high-performance data processing. For the latest information and licensing inquiries, please consult:

`http://www.getkerf.com`

Contents

1 Introduction

KeRF is a programming language built on pragmatism, borrowing ideas from many popular tools. The syntax of KeRF will be familiar enough to anyone who has programmed in C, Python or VBA. Data is described using syntax from JSON (JavaScript Object Notation), a text-based data interchange format. Queries to search, sort and aggregate data can be performed using syntax similar to SQL. KeRF's built-in commands have aliases which allow programmers to use names and terms they are already used to.

Beneath this friendly syntax, KeRF exposes powerful ideas inspired by the language APL and its descendants. APL has a well-earned reputation for extreme concision, and with practice you will find that KeRF similarly permits you to say a great deal with a few short words. Coming from other programming languages, you may be surprised by how much you can accomplish without writing loops, using conditional statements or declaring variables. KeRF provides a fluid interface between your intentions and your data.

1.1 Conventions

Throughout this manual, the names of functions and commands will be shown in a `monospaced font`. Transcripts of terminal sessions will be shown with sections typed by the user in blue:

```
KeRF> range 6
  [0, 1, 2, 3, 4, 5]
KeRF> sum(5, range 6)
  20
```

1.2 Using the REPL

A Read-Evaluate-Print Loop (REPL) is an interactive console session that allows you to type code and see results. The REPL is the main way you will be interacting with KeRF. If KeRF is in the current directory, you can start the REPL by typing `./kerf` and pressing return, and if if you have installed KeRF in your path, you can simply type `kerf`. The rest of this discussion will assume the latter case.

1.2.1 Command-Line Arguments

KeRF accepts several command-line flags to control its behavior. Throughout this manual we will be using the `-q` flag for some examples to avoid showing the KeRF startup logo for the sake of brevity.

Flag	Arguments	Behavior
-q	-	Suppress the startup banner.
-l	-	Enable debug logging.
-e	String Expression	Execute an expression.
-x	String Expression	Evaluate an expression and print the result.
-p	Port Number	Specify a listening port for starting an IPC server.

Summary of Command-Line Flags

The `-e` and `-x` flags differ by whether or not they display the result of a calculation. Either will exit the interpreter when complete:

```
> kerf -x "2+3"
5
> kerf -e "2+3"
>
```

If you provide a filename as a command-line argument, the contents of that file will be executed before opening the REPL. You may wish to conclude scripts with `exit(0)` so that they execute and then self-terminate:

```
> cat example.kerf
display join unfold range(10)
exit(0)
> kerf example.kerf
[0,
[0, 1],
[0, 1, 2],
[0, 1, 2, 3],
[0, 1, 2, 3, 4],
[0, 1, 2, 3, 4, 5],
[0, 1, 2, 3, 4, 5, 6],
[0, 1, 2, 3, 4, 5, 6, 7],
[0, 1, 2, 3, 4, 5, 6, 7, 8],
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
>
```

You could also accomplish the same by using `-x` and `load`:

```
> kerf -x "load 'example.kerf'"
```

1.2.2 The REPL

The REPL always begins with the `KeRF>` prompt. Type an expression, press return and the result will be printed, followed by an empty line. If an expression returns null, it will appear as an empty line. Trailing whitespace will generally be elided from REPL transcripts in this manual.

```
KeRF> 1+3 5 7
  [4, 6, 8]

KeRF> null

KeRF>
```

If you type several expressions separated by semicolons (`;`), each will be executed left to right and the value returned by the final expression will be printed. An empty expression returns null, so if you end a statement with a semicolon it will effectively suppress printing the result. Transcripts in this manual will often use this technique for the sake of brevity.

```
KeRF> one: 1; two: 2
  2
KeRF> one
  1
KeRF> a: 3 5 7;
KeRF>
```

If you type an expression with an unbalanced number of `[`, `(` or `{`, the REPL will prompt you with `>` to complete the expression on the next line. Remember, newlines and semicolons are always equivalent:

```
KeRF> [1 2 3
> 4 5 6]
  [[1, 2, 3],
   [4, 5, 6]]
```

1.3 Examples

The following are a few short KeRF programs to provide a taste of how the language can be employed.

Are two strings anagrams?

```
function are_anagrams(a, b) {
        return (sort a) match (sort b)
}
```

```
KeRF> are_anagrams("baton", "stick")
  0
KeRF> are_anagrams("setecastronomy","toomanysecrets")
  1
```

Run-length encode the elements of a list:

```
function rl_encode(x) {
        s:  null = mapback x                    // runs which contain the same value
        v:  enlist mapdown x[which not s]        // leading value for each run
        c:  1 + cdr count mapdown 0 explode s    // count of each run
        return {{v: v, c: c}}
}
```

```
KeRF> rt: rl_encode "AABCCCABBC"
```

v	c
A	2
B	1
C	3
A	1
B	2
C	1

And decode:

```
function rl_decode(x) {
        return flatten x["c"] take mapdown x["v"]
}
```

```
KeRF> rl_decode rt
  "AABCCCABBC"
```

Iteratively calculate terms of the Fibonacci sequence:

```
function fibs(n) {
        t: (n-1) {[x] last(x) join sum x} deconverge 1 1
        return first transpose t
}
```

```
KeRF> fibs 10
  [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

2 Installation

Pre-compiled evaluation copies of KeRF for Linux and OSX can be obtained from the project's public page on GitHub. Currently, evaluation copies expire periodically. KeRF is gaining new features frequently, so make sure you keep your installation up to date.

`https://github.com/kevinlawler/kerf`

2.1 Installing (Evaluation)

Fetch a local copy of the KeRF repository using `git clone`:

```
/Users/john/Desktop> git clone https://github.com/kevinlawler/kerf.git
Cloning into 'kerf'...
remote: Counting objects: 538, done.
remote: Total 538 (delta 0), reused 0 (delta 0), pack-reused 538
Receiving objects: 100% (538/538), 29.26 MiB | 1.65 MiB/s, done.
Resolving deltas: 100% (225/225), done.
Checking connectivity... done
/Users/john/Desktop> cd kerf/
/Users/john/Desktop/kerf>
```

The binaries located in the `/KerfREPL/linux` and `/KerfREPL/osx` directories are for 64-bit Linux and OSX, respectively. They are statically linked and include all library dependencies. Installation is as simple as placing the binary in a desired directory. Let's place it in a directory called `/opt/kerf` so that it is accessible by all users. It will be necessary to use the sudo command when creating this directory, as the base directory is owned by `root`:

```
/Users/john/Desktop> sudo mkdir /opt/kerf
Password:
/Users/john/Desktop> sudo cp KerfREPL/osx/kerf /opt/kerf/
/Users/john/Desktop> cd /opt/kerf
/opt/kerf> ls
kerf
```

You can then invoke it from the command line. The `-q` option (*quiet*) suppresses the KeRF logo at startup, for the purposes of brevity in these transcripts.

```
/opt/kerf> ./kerf -q
KeRF> 2+3
  5
KeRF> exit(0)
/opt/kerf>
```

From time to time, you should use the command `git pull` from within the directory you created via `git clone` to fetch the latest build of KeRF from this repository. Then simply repeat the above steps to move the fresh binary into its normal resting place.

2.2 Installing (Building from Source)

If you have been granted access to the KeRF source code, you can build your own binaries. From the base source directory, invoke `make clean` to remove any temporary or compiled files and then `make` to build a fresh set of binaries for your OS:

```
/Users/john/Desktop/kerf-source> make clean
find manual/ -type f -not -name '*.tex' | xargs rm
rm -f -r kerf kerf_test .//obj/*.o
/Users/john/Desktop/kerf-source> make
clang  -rdynamic -m64 -w -Os -c alter.c -o obj/alter.o

...

/Users/john/Desktop/kerf-source>
```

This process will produce a `kerf` executable in the source directory. You can then follow the steps described in the above section to place this in a directory accessible by other users or simply run it in place.

2.3 Adding KeRF to your Path

You may wish to add the KeRF binary to your PATH so that it can be accessed more easily. If you've placed the binary in the directory `/opt/kerf/`, edit `~/.profile` and add the following line:

```
export PATH=/opt/kerf/:$PATH
```

Open a fresh terminal or type `source ~/.profile` and you should now be able to invoke the `kerf` command from any directory.

3 Terminology

KeRF uses terminology from databases, statistics and array-oriented programming languages like APL. This section will serve as a primer for concepts which may seem unfamiliar.

3.1 Atomicity

Atomicity describes the manner in which values are *conformed* by particular *functions*.

A function which is not atomic will simply be applied to its arguments, behaving the same whether they are lists or atoms. `enlist` is not atomic:

```
KeRF> enlist 4
  [4]
KeRF> enlist [1, 9, 8]
  [[1, 9, 8]]
```

A unary function which is *atomic* will completely decompose any nested lists in the argument, operate on each atom separately, and then reassemble these results to match the shape of the original argument. Another way to think of this is that atomic functions "penetrate" to the atoms of their arguments. `not` is atomic:

```
KeRF> not 1
  0
KeRF> not [1, 0, 0, 1, 0]
  [0, 1, 1, 0, 1]
KeRF> not [1, 0, [1, 0], [0, 1], 0]
  [0, 1, [0, 1], [1, 0], 1]
```

Things get more interesting when dealing with *fully atomic* binary functions. The shapes of the arguments do not have to be identical, but they must recursively *conform*. Atoms conform with atoms. Lists conform with atoms and vice versa. Lists only conform with other lists if their lengths match and each successive pairing of their elements conforms. add is fully atomic:

```
KeRF> add(1, 2)
  3
KeRF> add(1 3 5, 10)
  [11, 13, 15]
KeRF> add(1 2 3, 4 5 6)
  [5, 7, 9]
```

3.2 Combinators

In the context of KeRF, a *combinator* is an operator which controls how a *function* is applied to *values*. Combinators express abstract patterns which recur frequently in programming. For example, consider the following loop:

```
function mysum(a) {
        s: 0
        for(i: 0; i < len(a); i: i+1) { s: add(s, a[i]) }
        return s
}
```

We're iterating over the indices of the list `x` from left to right, accumulating a result into the variable `s`. On each iteration, we take the previous `s` and combine it with the current element of `a` via the function `add`. This pattern is captured by the combinator `fold`, which takes a function as a left argument and a list as a right argument:

```
KeRF> add fold 37 15 4 8
  64
```

Think of `fold` as applying its function argument between the elements of its list argument:

```
KeRF> (((37 add 15) add 4) add 8)
  64
```

In this particular case we could have simply used the built-in function `sum`, but `fold` can be applied to any function- including functions you define yourself. Combinators are generally much more concise than writing explicit loops, and by virtue of having fewer "moving parts" avoid many classes of potential mistake entirely. If you use `fold` it isn't possible to have an "off-by-one" index error accessing elements of the list argument and several useful base cases are handled automatically. Familiarize yourself with all of KeRF's combinators- with practice, you may find you hardly ever need to use `for`, `do` and `while` loops at all!

KeRF understands the patterns combinators express and can sometimes perform dramatic optimizations when they are used in particular combinations with built-in functions or data with specific properties:

```
KeRF> timing(1);
KeRF> max fold range 50000
  49999
    0 ms
KeRF> {[a,b] max(a, b)} fold range 50000
  49999
    3.2 s
```

The former example allows KeRF to recognize the opportunity for short-circuiting `max fold` because the result of `range` is sorted. When `fold`ing a user-declared lambda, it must construct and then reduce the entire list.

3.3 Matrix

A *matrix* is a *vector* of *vector*s of uniform length and type. For example, the following is a matrix:

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

But this is not a matrix, because the rows are not of uniform length:

```
[[1, 2],
 [3, 4, 5, 6],
 [7, 8, 9]]
```

3.4 Valence

A function's *valence* is the number of arguments it takes. For example, `add` is a *binary function* which takes two arguments and thus has a valence of 2. `not`, on the other hand, is a *unary function* which takes a single argument and thus has a valence of 1. The term draws an analogy to linguistics and in turn chemistry, describing the way words and molecules form compounds.

3.5 Vector

A *vector* is a list of elements with a uniform type. If a list contains more than one type of element it is sometimes referred to as a *mixed-type list*. Vectors can store data more densely than mixed-type lists and as a result are often more efficient.

4 Datatypes

4.1 Numbers

KeRF has two numeric types: *integers* (or “ints”) and *floating-point numbers* (or “floats”). The results of numeric operations between ints and floats will coerce to floats, and some operators always yield floating-point results.

Integers consist of a sequence of digits, optionally preceded by `+` or `-`. They are internally represented as 64-bit signed integers and thus have a range of $-(2^{63})$ to $2^{63} - 1$.

```
42
010
+976
-9000
```

Integers can additionally be one of the special values `INF`, `-INF` or `NAN`, used for capturing arithmetic overflow and invalid elements of an integer vector:

```
KeRF> b: 10000000000000000000000000000000000000000
  INF
KeRF> kerf_type_name b
  "integer"
KeRF> 1 2 3[-5]
  NAN
```

Floats consist of a sequence of digits with an optional sign, decimal part and exponent. They are based on IEEE-754 double-precision 64-bit floats, and thus have a range of roughly $1.7 * 10^{\pm 308}$.

```
1.0
-379.8
-.2
.117e43
```

Floats can additionally be one of the special values `nan`, `-nan`, `inf` or `-inf`. `nan` has unusual properties in KeRF compared to most other languages. The behavior is intended to permit invalid results to propagate across calculations without disrupting other valid calculations:

```
KeRF> nan == nan
  1
KeRF> nan == -nan
  1
KeRF> 5/0
  inf
KeRF> 0/0
  nan
KeRF> -1/0
  -inf
```

4.2 Lists and Vectors

Lists are ordered containers of heterogenous elements. Lists have several literal forms. A sequence of numbers separated by whitespace is a valid list. This is an "APL-style" literal:

```
1 2 3 4
47 49
```

Alternatively, separate elements with commas (,) or semicolons (;) and enclose the list in square brackets for a more explicit "JSON-style" literal:

```
[1, 2, 3]
[4;5;6]
[42]
```

This manual will use both styles throughout the examples. Naturally, these styles can be nested together. The following examples are equivalent:

```
[1 2,3 4,5,6]
[[1, 2], [3, 4], 5, 6]
[[1;2], [3;4], 5, 6]
```

If a list consists entirely of items of the same type, it is a *vector*. Vectors can be represented more compactly than mixed-type lists and thus are more cache-friendly and provide better performance. KeRF has special optimizations for vectors of timestamps, characters, floats and integers.

Vector and list types each have their own special symbol for emptiness:

```
KeRF> 0 take 1 2 3
  INT[]
KeRF> 0 take 1.0 2 3
  FLOAT[]
KeRF> 0 take "ABC"
  ""
KeRF> 0 take [now(),now()]
  STAMP[]
KeRF> 0 take [1,"A"]
  []
```

4.3 Strings and Characters

KeRF character literals begin with a backtick (`) followed by double-quotes (") or single-quotes (') surrounding the character. Character literals which do not contain escape sequences may omit the quotation marks.

```
`A
`'B'
`"C"
```

KeRF supports all JSON string escape sequences, and additionally an escape for single-quotes:

```
`"\""       // double quote
`'\''       // single quote
`"\\"       // reverse solidus
`"\/"       // solidus
`"\b"       // backspace
`"\f"       // formfeed
`"\n"       // newline
`"\r"       // carriage return
`"\t"       // horizontal tab
`"\u0043"   // 4-digit unicode literal
```

The built in function char can convert a number into an equivalent character:

```
KeRF> char 65
  `"A"
```

Strings are lists of characters, and qualify as vectors. Strings are simply enclosed in double-quotes (") or single-quotes (').

```
"Hello, World!"
'goodbye,\ncruel world...'
```

4.4 Timestamps

Timestamps, or simply "stamps", are a flexible datatype which can represent dates, times, or a complete date-time. Times are internally represented in UTC at **nanosecond granularity**. Timestamp literals use a format similar to ISO-8601 except periods (`.`) are used as date field separators instead of dashes (`-`).

```
1997.07.16                // date only
19:20:30                  // time only
19:20:30.123              // time with milliseconds
1997.07.16T19:20:30       // datetime
1997.07.16T19:20:30.123   // datetime with milliseconds
```

Timestamps can be compared using the same operators as numeric types. The special builtin `stamp_diff` should be used for calculating the interval between timestamps:

```
KeRF> 2015.04.02 < 2015.05.01
  1
KeRF> stamp_diff(2015.05.03, 2015.05.01)
  172800000000000
```

KeRF provides special literals for relative date-times:

```
1y          // years
10m         // months
3d          // days
1h          // hours
2i          // minutes
1s          // seconds
```

These can also be combined as a single unit. For example,

```
KeRF> 2015.01.01 + 2m + 1d
  2015.03.02
KeRF> 2015.01.01 + 2m1d
  2015.03.02
KeRF> 2015.01.01 - 1h1i1s
  2014.12.31T22:58:59.000
```

Indexing is overloaded for timestamps to permit easy extraction of fields. The `date` and `time` fields produce a stamp and other fields produce an integer:

```
KeRF> d: now();
KeRF> d[["date", "time"]]
  [2015.11.14, 23:11:18.154]
KeRF> d[["month", "day", "hour", "minute"]]
  [11, 14, 23, 11]
KeRF> d[["year", "month", "day", "hour", "minute"]]
  [2015, 11, 14, 23, 11]
KeRF> d[["second", "millisecond", "nanosecond"]]
  [18, 154, 154858000]
```

4.5 Maps and Tables

A *Map* is an associative data structure which binds *keys* to *values*. KeRF maps are a generalization of JSON objects. Map literals are enclosed in a pair of curly braces ({ and }), and contain a series of comma delimited key-value pairs separated by a colon (:). JSON object syntax requires keys to be enclosed in double-quotes, but KeRF maps permit using single-quotes or bare identifiers.

```
{"a": 10, "b": 20}     // JSON style
{'a': 10, 'b': 20}     // optional single-quotes
{a: 10, b: 20}         // bare identifiers
```

A *Table* is a map in which each value is a list of equal length. Tables are enclosed in two pairs of curly braces ({{ and }}) and otherwise syntactically resemble maps. If non-list values are provided, they will be wrapped in lists. Values can also be omitted entirely to produce a table with empty columns.

```
{{a: 1 2, b: 3 4}}    // columns contain [1,2] and [3,4]
{{a: 1, b: 2}}        // columns contain [1] and [2]
{{a, b}}              // empty columns
```

If tables are serialized to JSON, KeRF will insert a key `is_json_table`- this permits tables to survive round-trip conversion:

```
KeRF> json_from_kerf {{a: 1 2, b: 3 4}}
  "{\"a\":[1,2],\"b\":[3,4],\"is_json_table\":[1]}"
```

The builtins `xkeys` and `xvals` can be used to extract a list of keys or values from a map or table:

```
KeRF> xkeys {a: 10 11 12, b: 20 21}
  ["a", "b"]
KeRF> xkeys {{a:1, b:2}}
  ["a", "b"]
KeRF> xvals {a: 10 11 12, b: 20 21}
  [[10, 11, 12], [20, 21]]
KeRF> xvals {{a:1, b:2}}
  [[1], [2]]
```

Many primitive operations penetrate to the values of maps and tables:

```
KeRF> 3 + {a: 10, b: 20}
  a:13, b:23
KeRF> 3 + {{a: 10, b: 20}}
```

a	b
13	23

4.6 Special Identifiers

KeRF uses reserved words to identify a number of special values.

The words `true` and `false` are boolean literals, equivalent to 1 and 0, respectively:

```
KeRF> true
  1
KeRF> false
  0
```

Inside a function, the words `self` or `this` may be used to refer to the current function. This is particularly useful for performing recursive calls in anonymous "lambda" functions:

```
KeRF> def foo(x) { return [this, self, x] }
  {[x] return [this, self, x]}
KeRF> foo(9)
  [{[x] return [this, self, x]}, {[x] return [this, self, x]}, 9]
```

The words `nil` or `null` may be used to refer to a null value:

```
KeRF> nil

KeRF> null

KeRF> kerf_type_name nil
  "null"
KeRF> kerf_type_name null
  "null"
```

The word `root` is a reference to KeRF's global scope. It contains all the variables which have been defined or referenced:

```
KeRF> root
  {}
KeRF> a: 24
  24
KeRF> b
  {}
KeRF> root
  {a:24, b:{}}
```

4.7 Index and Enum

Indexes and Enumerations are special lists which perform internal bookkeeping to improve the performance of certain operations.

An *Enumeration* performs *interning*. It keeps only one reference to each object and stores appearances as fixed-width indices. It is useful for storing repetitions of strings and lists, which cannot otherwise efficiently be stored as vectors. In all other respects an Enumeration appears to be a list. To create an Enumeration, use `hashed` or the unary `#` operator:

```
KeRF> a: hashed ["cherry", "peach", "cherry"]
  #["cherry", "peach", "cherry"]
KeRF> kerf_type_name a
  "enum"
```

Not only do Enumerations reduce the memory footprint of lists with a large number of repeated elements, they permit dramatically faster sorting. There is no benefit to making an Enumeration out of a vector of integers or floats.

```
KeRF> samples: {[x] rand(x, ["cherry", "peach", "zucchini"])};
KeRF> s1: samples(1000);
KeRF> s2: samples(10000);
KeRF> s3: samples(100000);
KeRF> timing 1
  1
KeRF> sort s1;
    2 ms
KeRF> sort s2;
    15 ms
KeRF> sort s3;
    134 ms
KeRF> sort hashed s1;
    0 ms
KeRF> sort hashed s2;
    4 ms
KeRF> sort hashed s3;
    28 ms
```

An *Index* is a list augmented with a B-Tree. This permits more efficient lookups and range queries. Do not use an index for data that will always be sorted in ascending order- KeRF tracks sorted lists internally. To create an Index, use `indexed` or the unary `=` operator:

```
KeRF> b: indexed 3 9 0 7
  =[3, 9, 0, 7]
KeRF> kerf_type_name b
  "sort"
```

5 Syntax

5.1 Expressions

Calling (or *applying*) a function in KeRF resembles most languages- use the name of the function followed by a parenthesized, comma-separated list of arguments:

```
KeRF> add(1, 2)
  3
```

If a function takes exactly one argument, the parentheses are optional. We call this style “prefix” function application:

```
KeRF> negate(3)
  -3
KeRF> negate 3
  -3
```

This syntax makes it easy to “chain” together a series of unary functions:

```
KeRF> last sort unique "ALPHABETICAL"
  `"T"
KeRF> last(sort(unique("ALPHABETICAL")))
  `"T"
```

If a function takes no arguments, you must remember to include parentheses- otherwise the function will be returned as a value instead of called:

```
KeRF> exit
  exit
KeRF> exit()
>
```

If a function takes exactly two arguments, it can be placed between the first and second argument as an “infix” operator:

```
KeRF> 1 add 2
  3
```

Many of the most frequently used functions have symbolic aliases. For example, `+` can be used instead of `add` and `*` can be used instead of `times`. There is no functional difference between the spelled-out names for these functions and the symbols.

```
KeRF> 3 * 5
  15
KeRF> *(3, 5)
  5
KeRF> 3 + 5
  8
KeRF> +(3, 5)
  5
```

Operators do not have precedence in KeRF. Expressions are evaluated strictly from right to left unless explicitly grouped with parentheses:

```
KeRF> 3 * 4 + 1
  15
KeRF> 4 + 1 * 3
  7
KeRF> (4 + 1) * 3
  15
```

KeRF's flexible syntax often provides many alternatives for writing the same expression. Select the arrangement that you feel is most clear. Adding parentheses to confusing-seeming expressions never hurts!

```
KeRF> 0.5 * 3**2
  4.5
KeRF> times(1/2, 3**2)
  4.5
KeRF> divide(1, 2) * exp(3, 2)
  4.5
KeRF> ((1 / 2) * (3 ** 2))
  4.5
```

<table>
<tr><th>Symbol</th><th>Unary Function</th><th>Binary Function</th></tr>
<tr><td>-</td><td>negate</td><td>minus</td></tr>
<tr><td>+</td><td>transpose</td><td>add</td></tr>
<tr><td>*</td><td>first</td><td>times</td></tr>
<tr><td>/</td><td>reverse</td><td>divide</td></tr>
<tr><td>|</td><td>len (length)</td><td>maxes/or</td></tr>
<tr><td>^</td><td>enumerate</td><td>take</td></tr>
<tr><td>%</td><td>distinct</td><td>mod (modulus)</td></tr>
<tr><td>&</td><td>part (partition)</td><td>mins/and</td></tr>
<tr><td>?</td><td>which</td><td>rand (random)</td></tr>
<tr><td>#</td><td>hashed</td><td>join</td></tr>
<tr><td>!</td><td>not</td><td>map (make map)</td></tr>
<tr><td>~</td><td>atom (is atom?)</td><td>match</td></tr>
<tr><td>=</td><td>indexed</td><td>equals</td></tr>
<tr><td><</td><td>ascend</td><td>less</td></tr>
<tr><td>></td><td>descend</td><td>greater</td></tr>
<tr><td>\</td><td></td><td>lsq</td></tr>
<tr><td><=</td><td></td><td>lesseq</td></tr>
<tr><td>>=</td><td></td><td>greatereq</td></tr>
<tr><td>==</td><td></td><td>equals</td></tr>
<tr><td>!=</td><td></td><td>noteq</td></tr>
<tr><td><></td><td></td><td>noteq</td></tr>
<tr><td>**</td><td></td><td>exp</td></tr>
<tr><td>_</td><td>floor</td><td></td></tr>
<tr><td>.</td><td>eval (evaluate)</td><td></td></tr>
<tr><td>:</td><td>ident (identity)</td><td></td></tr>
</table>

Symbolic Aliases of Built-in Functions

5.2 Indexing

KeRF has a uniform syntax for accessing elements of lists, maps and tables. Use square brackets to the right of a variable name or expression with an index or key to loop up:

```
KeRF> 3 7 15[1]
  7
KeRF> {a: 24, b: 29}["a"]
  24
```

If the provided index or key does not exist, indexing will return an appropriate type-specific null value, as provided by the `type_null` built-in function:

```
KeRF> 3 7 15[9]
  NAN
KeRF> 3 7 15[-1]
  NAN
KeRF> "ABC"[9]
  `" "
```

Floating-point indices to lists will be truncated, for convenience:

```
KeRF> 3 7 15[0.5]
  3
KeRF> 3 7 15[1.6]
  7
```

Indexing is right-atomic. If the indices are a list, the indexing operation will accumulate a list of results:

```
KeRF> "ABC"[2 1 0 0 2 3 0 0 1]
  "CBAAC AAB"
KeRF> 34 19 55 32[0 1 0 1 2 2]
  [34, 19, 34, 19, 55, 55]
```

One application of this type of collective indexing is the basis of `sort`:

```
KeRF> a: 27 15 9 55 0
  [27, 15, 9, 55, 0]
KeRF> a[ascend a]
  [0, 9, 15, 27, 55]
```

The shape of the result of indexing will always match the shape of the indices. Consider this example, where we index a list with a 2x2 matrix and get back a 2x2 matrix:

```
KeRF> 11 22 33 44[[0 1, 2 3]]
  [[11, 22],
   [33, 44]]
```

Indexing a particular element from a multidimensional structure requires several indexing operations:

```
KeRF> [11 22, 33 44][0][1]
  33
```

5.3 Assignment

KeRF uses the colon (:) as an assignment operator, unlike the convention of "=" from many other programming languages. SQL uses = as a comparison operator, JSON uses : as an assignment operator in map literals and KeRF syntax attempts to be a superset of both JSON and SQL.

Values may be assigned to variables with : and retrieved by using the variable name. Note that uninitialized variables behave as containing an empty map:

```
KeRF> a
  {}
KeRF> a: 3 7 19
  [3, 7, 19]
KeRF> a
  [3, 7, 19]
KeRF> a[1]
  7
```

Assignment may be combined with indexing to assign to specific cells of a list or keys of a map:

```
KeRF> a: range 4
  [0, 1, 2, 3]
KeRF> a[1]: 99
  [0, 99, 2, 3]
KeRF> a
  [0, 99, 2, 3]
KeRF> b: {bravo: 3, tango: 6};
KeRF> b["bravo"]: 99
  {bravo:99, tango:6}
```

Note KeRF's copy-on-write semantics:

```
KeRF> a: 0 1 2 3;
KeRF> b: a
  [0, 1, 2, 3]
KeRF> b[1]:99
  [0, 99, 2, 3]
KeRF> a
  [0, 1, 2, 3]
```

As with indexing, it is possible to perform collective "spread" assignment:

```
KeRF> a: 8 take 0
  [0, 0, 0, 0, 0, 0, 0, 0]
KeRF> a[1 2 5]:99
  [0, 99, 99, 0, 0, 99, 0, 0]
KeRF> b: 8 take 0
  [0, 0, 0, 0, 0, 0, 0, 0]
KeRF> b[1 2 5]:11 22 33
  [0, 11, 22, 0, 0, 33, 0, 0]
```

It is also possible to perform compound assignment, treating `:` like a combinator which takes a binary function as a left argument and applies the old value and the right argument to this function before performing the assignment:

```
KeRF> a: 0 0 0
  [0, 0, 0]
KeRF> a#: 99
  [0, 0, 0, 99]
KeRF> a join: 47
  [0, 0, 0, 99, 47]
```

Compound assignment can be combined with indexing. Be warned, this can get confusing fairly quickly:

```
KeRF> a: 0 0 0
  [0, 0, 0]
KeRF> a[1]+:4
  [0, 4, 0]
KeRF> a[0 2]+:1
  [1, 4, 1]
KeRF> a[0 2]#:55
  [[1, 55], 4, [1, 55]]
KeRF> a[0 2]#:3 4
  [[1, 55, 3], 4, [1, 55, 4]]
```

To modify an element of a multidimensional structure, use multiple indexing expressions:

```
KeRF> a:[11 22, 33 44]
  [[11, 22],
   [33, 44]]
KeRF> a[0][1]:99
  [[11, 99],
   [33, 44]]
KeRF> a[0 1][0]#:0
  [[[11, 0], 99],
   [[33, 0], 44]]
```

5.4 Control Structures

KeRF has a familiar, simple set of general-purpose control structures. Note that parentheses and curly braces are *never* optional for control structures.

5.4.1 Conditionals

KeRF has a C-style `if` statement with optional `else if` and `else` clauses. Like the C ternary operator (`?:`), KeRF `if` statements can be used as part of an expression. Each curly-bracketed clause returns the value of its last expression.

```
KeRF> if (2 < 3) { 25 } else { 32 }
  25
KeRF> if (2 > 3) { 25 } else { 32 }
  32
```

Be careful using conditionals across multiple lines- not all coding styles are equal! In KeRF, newlines are statement separators. The following indentation style will work the way you expect:

```
if (a <  b) {
        c : 100
} else if (a == b) {
        c : 200
} else {
        c : 400
}
```

While this style will have surprising behavior. Implied statement separators shown in red:

```
if (a <  b) {;
        c : 100;
};
else if (a == b) {;
        c : 200;
};
else {;
        c : 400;
}
```

This style will also have surprising behavior:

```
if      (a <  b) { c : 100 };
else if (a == b) { c : 200 };
else             { c : 400 };
```

For simple operations, favor placing your conditionals on a single line. For complex operations, favor the first indentation style above.

5.4.2 Loops

KeRF provides a C-style `for` loop. The header consists of an initialization expression, a predicate and an updating expression. Note that the `for` loop itself returns null:

```
KeRF> for (i: 0; i < 4; i: i+1) { display 2*i }
0
2
4
6

KeRF>
```

KeRF also provides a C-style `while` loop. Note how in this example the loop returns its final calculation:

```
KeRF> t: 500; while(t > 32) { display t; t: floor t/2 }
500
250
125
62
  31

KeRF>
```

If you simply want to repeat an expression a fixed number of times, use `do`:

```
KeRF> do (3) { display 42 }
42
42
42

KeRF>
```

KeRF loops do **not** provide C-style `break` and `continue`. Instead, break your program into functions and use `return` when you wish to prematurely exit a loop.

Combinators and built-in functions can be substituted for loops in many situations:

```
KeRF> display mapdown range(0, 8, 2);
0
2
4
6

KeRF> {[t] t>32 } {[t] display t; floor t/2 } converge 500
500
250
125
62
  31

KeRF> display mapdown take(3, 42);
42
42
42
```

5.4.3 Function Declarations

Functions can be declared using the `function` or `def` keywords and providing a parenthesized argument list. The final statement in a function body will be implicitly returned, and at any point in a function body you can instead use the `return` keyword to explicitly return.

```
function is_even(n) {
        return (n % 2) == 0
}

def divisible(a, b) {
        return (a % b) == 0
}
```

It is also possible to define anonymous functions as part of an expression. Some languages refer to these as *lambdas*, in reference to the lambda calculus, a formal model of computation based on the manipulation of anonymous functions. Anonymous functions are enclosed in curly brackets and may provide a square-bracketed ([and]) argument list:

```
KeRF> {[a, b] 2*a+b }(3, 5)
  16
```

Storing a lambda in a variable is precisely equivalent to defining a function with `function` or `def`.

```
KeRF> divisible: {[a, b] (a % b) == 0 }
  {[a, b] (a % b) == 0}
KeRF> divisible(6, 3)
  1
KeRF> divisible(7, 2)
  0
```

KeRF uses *lexical scope*. This means that when variables are referenced, the *definition textually closest* to the reference will be used. The following example will print 25, because `inner` captures the definition of `x` in `outer_1` when it is created. The definition of `x` in `outer_2` is not used when this function is evaluated, nor is the top-level definition of `x`.

```
x: 35
function outer_1() {
        x: 25
        function inner() {
                return x
        }
        return inner;
}
function outer_2() {
        x: 15
        l: outer_1()
        return l()
}
display outer_2()
```

6 SQL

KeRF understands a subset of SQL (Structured Queried Language), a popular programmatic interface for relational databases. You can blend SQL-style queries with imperative statements and access the full range of KeRF predicates and logical operators while filtering and selecting results.

SQL keywords are not case-sensitive. To distinguish SQL keywords, the following examples will use uppercase exclusively. When describing the syntax of SQL statements, sections which can contain table names, field names or other types of subexpressions will be shown in **bold**. If a section is optional, it will be enclosed in bold square brackets (**[** and **]**). For example:

```
SELECT fields [ AS name ] FROM table ...
```

6.1 INSERT

`INSERT` is the simplest type of SQL statement. It is used for creating or appending to tables. `INSERT` can perform single or bulk insertions, the latter of which is much more efficient.

```
INSERT INTO table VALUES data
```

table can be a table literal or the name of a variable containing a table. In the latter case, the table will be modified in-place. **data** can be a list, matrix, map or table.

A single insertion can take values from a comma-separated, parenthesized list- this is special SQL syntax. Alternatively, use an ordinary square-bracketed KeRF list. You can also use a map, which more explicitly shows column names in the source data.

```
KeRF> INSERT INTO {{name, email, level}} VALUES ("bob", "b@ob.com", 7)
```

name	email	level
bob	b@ob.com	7

```
KeRF> INSERT INTO {{name, email, level}} VALUES ["bob", "b@ob.com", 7]
```

name	email	level
bob	b@ob.com	7

```
KeRF> INSERT INTO {{name, email, level}} VALUES {name: "bob", level: 7, email: "b@ob.com"}
```

name	email	level
bob	b@ob.com	7

Note that a map which is to be inserted must have **exactly** the same key set as the destination table:

```
KeRF> INSERT INTO {{name, email, level}} VALUES {name: "bob", level: 7}

  INSERT INTO {{name, email...
  ^
 Length error
KeRF> INSERT INTO {{name, email, level}} VALUES {name: "bob", level: 7, hobbies: "needlepoint"}

  INSERT INTO {{name, email...
  ^
 Column error
```

Bulk insertions require either a matrix or a table:

```
KeRF> employees: [["bob","alice","jerry"]
                  ["b@ob.com", "alice@gmail.com", "jerry@zombo.com"]
                  [7, 9, 43]];
KeRF> INSERT INTO {{name, email, level}} VALUES employees
```

name	email	level
bob	b@ob.com	7
alice	alice@gmail.com	9
jerry	jerry@zombo.com	43

An empty table will accept an INSERT from any map or table. List or matrix elements will be assigned default column names:

```
KeRF> INSERT INTO {{}} VALUES {legume: "Black Bean", dish: "Casserole"}
```

legume	dish
Black Bean	Casserole

```
KeRF> INSERT INTO {{}} VALUES employees
```

col	col1	col2
bob	b@ob.com	7
alice	alice@gmail.com	9
jerry	jerry@zombo.com	43

6.2 DELETE

DELETE is used for removing rows from a table. KeRF's columnar representation of tables means that a DELETE runs in linear time with respect to the number of rows in the table. Keep this in mind, and avoid repeated DELETEs over large (on-disk) datasets.

```
DELETE FROM table [ WHERE condition ]
```

table can be a table literal or the name of a variable containing a table. In the latter case, the table will be modified in-place. **condition** can be any KeRF expression, using the names of columns from **table** as variables. For more information about WHERE, see the discussion of SELECT.

If provided a reference to a table stored in a variable, DELETE will return the name of that variable. Otherwise, it will return the modified table itself:

```
KeRF> t: {{a:range(5000), b:rand(5000, 100.0)}}

┌─┬───────┐
│a│b      │
├─┼───────┤
│0│16.4771│
│1│27.3974│
│2│28.3558│
│3│12.2126│
│4│45.1148│
│5│81.5326│
│6│ 95.726│
│7│38.1769│
│.│     ..│
└─┴───────┘

KeRF> count t
  5000
KeRF> DELETE FROM t WHERE b between [0, 50]
  "t"
KeRF> count t
  2474
KeRF> DELETE FROM {{a:range(10), b:rand(10, 100.0)}} WHERE (a%2) = 1

┌─┬───────┐
│a│b      │
├─┼───────┤
│0│75.3017│
│2│   67.3│
│4│19.4571│
│6│66.3412│
│8│ 66.116│
└─┴───────┘
```

As you might expect, if you don't use a WHERE clause, DELETE will remove all the rows of a table:

```
KeRF> DELETE FROM t
  "t"
KeRF> t

┌─┬─┐
│a│b│
├─┼─┤
│ │ │
└─┴─┘
```

6.3 SELECT

SELECT performs queries. It can be used to extract, aggregate or transform the contents of tables, producing new tables.

```
SELECT fields [ AS name ] FROM table
        [ WHERE condition ]
        [ GROUP BY aggregate ]
```

In its simplest form, SELECT can be used to slice a desired set of columns out of a table:

```
KeRF> people
```

name	age	gender	job
Hamilton Butters	37	M	Janitor
Emma Peel	29	F	Secret Agent
Jacques Maloney	48	M	Private Investigator
Renee Smithee	31	F	Programmer
Karen Milgram	16	F	Student
Chuck Manwich	29	M	Janitor
Steak Manhattan	18	M	Secret Agent
Tricia McMillen	29	F	Mathematician

```
KeRF> SELECT name, gender FROM people
```

name	gender
Hamilton Butters	M
Emma Peel	F
Jacques Maloney	M
Renee Smithee	F
Karen Milgram	F
Chuck Manwich	M
Steak Manhattan	M
Tricia McMillen	F

Selected columns can be renamed by specifying an AS clause for each:

```
KeRF> SELECT age AS person_age, gender AS sex FROM people
```

person_age	sex
37	M
29	F
48	M
31	F
16	F
29	M
18	M
29	F

The wildcard * can be used to refer to all columns in a table. It is also possible to use a variety of collective functions like count or avg when selecting columns. When calculated columns are not given a name explicitly via AS, a default name will be supplied.

```
KeRF> SELECT count(*), sum(age), avg(age) AS average_age FROM people
```

col	age	average_age
8	237	29.625

It is possible to reference user-defined functions in a SELECT, but if they are not atomic you may need to explicitly apply them to elements of a column using combinators:

```
KeRF> revname: {[x] p:explode(`" ",x); implode(", ", reverse p)};
KeRF> from_dogyears: {[x] if (x < 21) { x/10.5 } else { x/4 }};
KeRF> SELECT revname mapdown name, from_dogyears mapdown age AS dog_age FROM people
```

name	dog_age
Butters, Hamilton	9.25
Peel, Emma	7.25
Maloney, Jacques	12.0
Smithee, Renee	7.75
Milgram, Karen	1.52381
Manwich, Chuck	7.25
Manhattan, Steak	1.71429
McMillen, Tricia	7.25

6.3.1 WHERE

The WHERE clause permits filtering of results. Only rows which adhere to the contraints given as **condition** will be returned:

```
KeRF> SELECT * FROM people WHERE age > 30
```

name	age	gender	job
Hamilton Butters	37	M	Janitor
Jacques Maloney	48	M	Private Investigator
Renee Smithee	31	F	Programmer

You can form a *conjunction* with several conditions by separating them with commas. Given a conjunction, the result is only selected if all conditions are satisfied. This is equivalent to performing a logical *AND*:

```
KeRF> SELECT * FROM people WHERE gender = "M", age > 30
```

name	age	gender	job
Hamilton Butters	37	M	Janitor
Jacques Maloney	48	M	Private Investigator

You can also form a conjuction by using the KeRF and operator, but in this case you *must* parenthesize subexpressions. Remember: KeRF evaluates expressions right to left unless otherwise parenthesized, so a = b and c > d is equivalent to a = (b and (c > d)):

```
KeRF> SELECT * FROM people WHERE gender = "M" and age > 30

   gender = "M" and age > 30
                    ^
 Type error

KeRF> SELECT * FROM people WHERE (gender = "M") and (age > 30)
```

name	age	gender	job
Hamilton Butters	37	M	Janitor
Jacques Maloney	48	M	Private Investigator

6.3.2 GROUP BY

The `GROUP BY` clause can be used to gather together sets of rows which match on a particular column. In its simplest form, it behaves somewhat like the built-in function `part`. Note that the grouped-by column is included in the results:

```
KeRF> SELECT name, age FROM people GROUP BY job
```

job	name	age
Janitor	[Hamilton Butters, Chuck Manwich]	[37, 29]
Secret Agent	[Emma Peel, Steak Manhattan]	[29, 18]
Private Investigator	[Jacques Maloney]	[48]
Programmer	[Renee Smithee]	[31]
Student	[Karen Milgram]	[16]
Mathematician	[Tricia McMillen]	[29]

You may often want to use collective functions to reduce each list of results:

```
KeRF> SELECT count(name) AS num, avg(age) FROM people GROUP BY job
```

job	num	age
Janitor	2	33.0
Secret Agent	2	23.5
Private Investigator	1	48.0
Programmer	1	31.0
Student	1	16.0
Mathematician	1	29.0

6.4 Joins

KeRF provides built-in functions `left_join` and `asof_join` which can be used to align and combine tables:

```
KeRF> livesin
```

name	nationality
Hamilton Butters	USA
Emma Peel	UK
Jacques Maloney	France
Renee Smithee	France
Karen Milgram	USA
Chuck Manwich	Canada
Tricia McMillen	UK

```
KeRF> SELECT name, age, nationality FROM left_join(people, livesin, "name")
```

name	age	nationality
Hamilton Butters	37	USA
Emma Peel	29	UK
Jacques Maloney	48	France
Renee Smithee	31	France
Karen Milgram	16	USA
Chuck Manwich	29	Canada
Steak Manhattan	18	null
Tricia McMillen	29	UK

6.5 Limiting

If you wish to retrieve the first n items of a query, as in a SQL `LIMIT` clause, you can use the built-in function `first`:

```
KeRF> first(2, select name, age from people where gender = "F")
```

name	age
Emma Peel	29
Renee Smithee	31

But beware- if the result has fewer than n rows, this approach will replicate them. A better approach is to define a new function which takes the minimum of n and the length of the result:

```
KeRF> first(2, select name, age from people where name = "Emma Peel")
```

name	age
Emma Peel	29
Emma Peel	29

```
KeRF> limit: {[n, t] first(min(count t, n), t)};
KeRF> limit(2, select name, age from people where name = "Emma Peel")
```

name	age
Emma Peel	29

6.6 Ordering

If you wish to sort tables along a column, as in a SQL `ORDER BY` clause, you can use the built-in functions `ascend` or `descend` along with indexing:

```
KeRF> people[ascend select job from people]
```

name	age	gender	job
Hamilton Butters	37	M	Janitor
Chuck Manwich	29	M	Janitor
Tricia McMillen	29	F	Mathematician
Jacques Maloney	48	M	Private Investigator
Renee Smithee	31	F	Programmer
Emma Peel	29	F	Secret Agent
Steak Manhattan	18	M	Secret Agent
Karen Milgram	16	F	Student

6.7 Performance

`WHERE` clauses have a special understanding of certain KeRF verbs and can achieve significant performance boosts in the right circumstances.

```
KeRF> n: 200000;
KeRF> i: range(n);
KeRF> v: rand(n, 100.0);
KeRF> iv: indexed v;
KeRF> find: {[x] select count(*) from {{i:i, v:x}} where v between [0, 10]};

KeRF> timing 1;
KeRF> find v;
    16 ms
KeRF> find iv;
    10 ms
```

For best results, order `WHERE` conjunctions to perform the largest reduction of data first, or take advantage of `indexed` or `enum` columns as early as possible:

```
KeRF> n: 200000;
KeRF> t: {{a: range(n), b: rand(n, 100.0), c: rand(n, 6)}}
```

a	b	c
0	82.268	2
1	80.5227	0
2	13.5797	1
3	80.2291	3
4	61.5329	3
5	67.2546	1
6	64.5684	5
7	29.7027	2
.	..	.

```
KeRF> timing 1;
KeRF> select * from t where b > 50, c = 1;
    25 ms
KeRF> select * from t where c = 1, b > 50;
    13 ms
KeRF> select * from t where (c = 1) and (b > 50);
    14 ms
```

7 Built-In Function Reference

7.1 abs - Absolute Value

`abs(x)`

Calculate the absolute value of x. Atomic.

```
KeRF> abs -4 7 -2.19 NaN
  [4, 7, 2.19, nan]
```

7.2 acos - Arc Cosine

`acos(x)`

Calculate the arc cosine (inverse cosine) of x, expressed in radians, within the interval [-1,1]. Atomic. The results of acos will always be floating point values.

```
KeRF> acos 0.5 -0.2 1
  [1.0472, 1.77215, 0]
KeRF> cos(acos 0.5 -0.2 1 4)
  [0.5, -0.2, 1, nan]
```

7.3 add - Add

`add(x, y)`

Calculate the sum of x and y. Fully atomic.

```
KeRF> add(3, 5)
  8
KeRF> add(3, 9 15 -7)
  [12, 18, -4]
KeRF> add(9 15 -7, 3)
  [12, 18, -4]
KeRF> add(9 15 -7, 1 3 5)
  [10, 18, -2]
```

The symbol + is equivalent to add when used as a binary operator:

```
KeRF> 2 4 3+9
  [11, 13, 12]
```

7.4 and - Logical AND

`and(x, y)`

Calculate the logical *AND* of x and y. This operation is equivalent to the primitive function min. Fully atomic.

```
KeRF> and(1 1 0 0, 1 0 1 0)
  [1, 0, 0, 0]
KeRF> and(1 2 3 4, 0 -4 9 0)
  [0, -4, 3, 0]
```

The symbol & is equivalent to and when used as a binary operator:

```
KeRF> 1 1 0 0 & 1 0 1 0
  [1, 0, 0, 0]
```

7.5 ascend - Ascending Indices

ascend(x)

For a list x, generate a list of indices into x in ascending order of the values of x.

```
KeRF> t:5 2 3 1
  [5, 2, 3, 1]
KeRF> ascend t
  [3, 1, 2, 0]
KeRF> t[ascend t]
  [1, 2, 3, 5]
```

Strings are sorted in lexicographic order:

```
KeRF> ascend ["Orange","Apple","Pear","Aardvark","A"]
  [4, 3, 1, 0, 2]
```

When applied to a map, ascend will sort the keys by their values and produce a list:

```
KeRF> ascend {"A":2, "B":9, "C":0}
  ["C", "A", "B"]
```

The symbol < is equivalent to ascend when used as a unary operator:

```
KeRF> <5 2 3 1
  [3, 1, 2, 0]
```

7.6 asin - Arc Sine

asin(x)

Calculate the arc sine (inverse sine) of x, expressed in radians, within the interval [-1,1]. Atomic. The results of asin will always be floating point values.

```
KeRF> asin 0.5 -0.2 1
  [0.523599, -0.201358, 1.5708]
KeRF> sin(asin 0.5 -0.2 1 4)
  [0.5, -0.2, 1, nan]
```

7.7 asof_join - Asof Join

asof_join(x, y, k1, k2)

Perform a "fuzzy" left join. Behaves as left_join for the first three arguments. k2 is a string, list or map indicating columns which will match if the values in y are less than or equal to x. Often this operation is applied to timestamp columns, but it works for any other comparable column type.

```
KeRF> t: {{a: 1 2 2 3, b: 10 20 30 40}}
```

a	b
1	10
2	20
2	30
3	40

```
KeRF> u: {{b: 19 17 32 8, c: ["A","B","C","D"]}}
```

b	c
19	A
17	B
32	C
8	D

```
KeRF> asof_join(t, u, [], "b")
```

a	b	c
1	10	D
2	20	A
2	30	A
3	40	C

7.8 atan - Arc Tangent

atan(x)

Calculate the arc tangent (inverse tangent) of x, expressed in radians. Atomic. The results of atan will always be floating point values.

```
KeRF> atan 0.5 -0.2 1 4
  [0.463648, -0.197396, 0.785398, 1.32582]
KeRF> tan(atan 0.5 -0.2 1 4)
  [0.5, -0.2, 1, 4]
```

7.9 atlas - Atlas Of

atlas(x)

Create an atlas from a map. An atlas is the schemaless NoSQL equivalent of a table. Atlases are automatically indexed in such a way that all key-queries are indexed.

```
KeRF> atlas({name:["bob", "alice", "oscar"], id:[123, 421, 233]})
  atlas[{name:["bob", "alice", "oscar"], id:[123, 421, 233]}]
```

7.10 atom - Is Atom?

atom(x)

A predicate which returns 0 if `x` is a list, and 1 if `x` is a non-list (atomic) value.

```
KeRF> atom `"A"
  1
KeRF> atom "A String"
  0
KeRF> atom 37
  1
KeRF> atom -0.2
  1
KeRF> atom 2015.03.31
  1
KeRF> atom null
  1
KeRF> atom [2, 5, 16]
  0
KeRF> atom a: 45, b: 76
  1
```

7.11 avg - Average

`avg(x)`

Calculate the arithmetic mean of the elements of a list `x`. Equivalent to `(sum x)/count_nonnull x`.

```
KeRF> avg 3 7 12.5 9
  7.875
```

7.12 between - Between?

`between(x, y)`

Predicate which returns 1 if `x` is between the first two elements of the list `y`. Equivalent to `(x >= y[0]) & (x <= y[1])`.

```
KeRF> between(2 5 17, 3 10)
  [0, 1, 0]
```

Be careful- `between` will always fail if `y` is not a list or does not have the correct length:

```
KeRF> between(2 5 17, 3)
  [0, 0, 0]

KeRF> 3[1]
  NAN
```

7.13 btree - BTree

`btree(x)`

Equivalent to `indexed(x)`.

7.14 bucketed - Bucket Values

`bucketed(x, y)`

Equivalent to `floor (<<y) * x / count y`.

7.15 car - Contents of Address Register

`car(x)`

Select the first element of the list `x`. Atomic types are unaffected by this operation. Equivalent to `first(x)`. `car` is a reference to the Lisp primitive of the same name, which selected the first element of a pair.

```
KeRF> car 32 83 90
  32
KeRF> car 409
  409
KeRF> nil = car []
  1
```

7.16 cdr - Contents of Data Register

`cdr(x)`

Select all the elements of the list `x` except for the first. Atomic types are unaffected by this operation. Equivalent to `drop(1, x)`. `cdr` is a reference to the Lisp primitive of the same name, which selected the second element of a pair.

```
KeRF> cdr 32 83 90
  [83, 90]
KeRF> cdr 409
  409
```

7.17 ceil - Ceiling

`ceil(x)`

Compute the smallest integer following a number `x`. Atomic.

```
KeRF> ceil -3.2 0.4 0.9 1.1
  [-3, 1, 1, 2]
```

Taking the ceiling of a string or char converts it to uppercase:

```
KeRF> ceil "Hello, World!"
  "HELLO, WORLD!"
```

7.18 char - Cast to Char

`char(x)`

Cast a number or list `x` to a char or string, respectively.

```
KeRF> char 65
  `"A"
KeRF> char 66.7
  `"B"
KeRF> char 72 101 108 108 111 44 32 75 101 82 70 33
  "Hello, KeRF!"
```

7.19 close_socket - Close Socket

`close_socket(x)`

Given a socket handle as obtained with `open_socket`, close the connection.

7.20 cos - Cosine

`cos(x)`

Calculate the cosine of `x`, expressed in radians. Atomic. The results of `cos` will always be floating point values.

```
KeRF> cos 3.14159 1 -20
  [-1, 0.540302, 0.408082]
KeRF> acos cos 3.14159 1 -20
  [3.14159, 1, 1.15044]
```

7.21 cosh - Hyperbolic Cosine

`cosh(x)`

Calculate the hyperbolic cosine of `x`, expressed in radians. Atomic. The results of `cosh` will always be floating point values.

```
KeRF> cosh 3.14159 1 -20
  [11.5919, 1.54308, 2.42583e+08]
```

7.22 count - Count

`count(x)`

Determine the number of elements in `x`. Equivalent to `len(x)`. Atomic elements have a count of 1.

```
KeRF> count 4 7 9
  3
KeRF> count [4 7 9, 23 32]
  2
KeRF> count 5
  1
KeRF> count {a:23, b:45}
  1
```

7.23 count_nonnull - Count Non-Nulls

`count_nonnull(x)`

Determine the number of elements in `x` which are not null. Equivalent to `sum not isnull x`.

```
KeRF> count_nonnull 1 2 3
  3
KeRF> count_nonnull [nan, null, 45]
  1
```

7.24 count_null - Count Nulls

`count_null(x)`

Determine the number of elements in `x` which are null. Equivalent to `sum isnull x`.

```
KeRF> count_null 1 2 3
  0
KeRF> count_null [nan, null, 45]
  2
```

7.25 descend - Descending Indices

`descend(x)`

For a list `x`, generate a list of indices into `x` in descending order of the values of `x`.

```
KeRF> t:5 2 3 1
  [5, 2, 3, 1]
KeRF> descend t
  [0, 2, 1, 3]
KeRF> t[descend t]
  [5, 3, 2, 1]
```

Strings are sorted in lexicographic order:

```
KeRF> descend ["Orange","Apple","Pear","Aardvark","A"]
  [2, 0, 1, 3, 4]
```

When applied to a map, `descend` will sort the keys by their values and produce a list:

```
KeRF> ascend {"A":2, "B":9, "C":0}
  ["B", "A", "C"]
```

The symbol > is equivalent to `descend` when used as a unary operator:

```
KeRF> >5 2 3 1
  [0, 2, 1, 3]
```

7.26 dir_ls - Directory Listing

```
dir_ls(path)
dir_ls(path, full)
```

List the files and directories at a filesystem `path`. If `full` is provided and truthy, list complete paths to the elements of the directory. Otherwise, list only the base names.

```
KeRF> dir_ls("/Users/john/Sites")
  [".DS_Store", ".localized", "images", "index.html", "subforum.php"]
KeRF> dir_ls("/Users/john/Sites", 1)
  ["\/Users\/john\/Sites\/.DS_Store"
   "\/Users\/john\/Sites\/.localized"
   "\/Users\/john\/Sites\/images"
   "\/Users\/john\/Sites\/index.html"
   "\/Users\/john\/Sites\/subforum.php"]
```

7.27 display - Display

```
display(x)
```

Print a display representation of data to stdout. Equivalent to `out join(rep x, '\n')`. A key difference between calling this function from the Repl and using the Repl's natural value printing is that `display` will print the entire result:

```
KeRF> range 50
  [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, 32, 33, 34, 35, 36, 37,
38, 39, 40, ...]
KeRF> display range 50
[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, 32, 33, 34, 35, 36, 37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

7.28 distinct - Distinct Values

```
distinct(x)
```

Select the first instance of each item in a list `x`. Atomic types are unaffected by this operation.

```
KeRF> distinct "BANANA"
  "BAN"
KeRF> distinct 2 3 3 5 3 4 5
  [2, 3, 5, 4]
```

The symbol % is equivalent to `distinct` when used as a unary operator:

```
KeRF> %"BANANA"
  "BAN"
```

7.29 `divide` - Divide

`divide(x, y)`

Divide `x` by `y`. Fully atomic. The results of `divide` will always be floating point values.

```
KeRF> divide(3, 5)
  0.6
KeRF> divide(-1, 0)
  -inf
KeRF> divide(3, 2 4 5 0)
  [1.5, 0.75, 0.6, inf]
KeRF> divide(1 3 4 0, 9)
  [0.111111, 0.333333, 0.444444, 0.0]
KeRF> divide(10 5 3, 7 9 3)
  [1.42857, 0.555556, 1.0]
```

The symbol / is equivalent to `divide` when used as a binary operator:

```
KeRF> 10 5 3 / 7 9 3
  [1.42857, 0.555556, 1.0]
```

7.30 `dlload` - Dynamic Library Load

`dlload(filename, function, argcount)`

Load a dynamic library function and return a KeRF function which can be invoked to call into it. `filename` is the name of the library, `function` is the name of the function and `argcount` is the number of arguments the function takes.

Let's look at a very simple of writing a C function which can be called from KeRF. Begin with the following saved as `dynamic.c`. All values passed into and out of dynamic libraries are `KERF` structures, as defined below.

```
#include <stdint.h>
typedef struct kerf0{
        char m; char a; char h; char t; int32_t r;
        union{
                int64_t i; double f; char c; char* s; struct kerf0*k;
                struct { int64_t n; char g[]; };
        };
} *KERF, KERF0;

#include <stdio.h>
KERF foreign_function_example(KERF argument) {
        int64_t value = argument->i;
        printf("Hello from C! You gave me a %lld.\n", value);
        return 0;
}
```

To compile on OSX, invoke your system's C compiler as follows:

```
cc -m64 -dynamiclib dynamic.c -o example.dylib
```

Then, with the resulting `example.dylib` in the current directory, load it from KeRF:

```
KeRF> f: dlload("example.dylib", "foreign_function_example", 1)
  {OBJECT:foreign_function_example}
KeRF> f(42)
Hello from C! You gave me a 42.
```

To return a result from a dynamic library, you must `malloc` your own `KERF` structs and initialize the fields appropriately. KeRF will automatically transfer `malloc`ed foreign function return values into its own reference-counted memory pool. Here's a very simple C function which constructs a KeRF integer value:

```
#include <stdlib.h>

KERF kerf_int(int64_t value) {
        KERF k = malloc(sizeof(KERF0));
        k -> m = 0;
        k -> h = 0;
        k -> a = 0;     // attribute flags
        k -> r = -1;    // reference count
        k -> n = 0;     // number of elements
        k -> t = 2;     // type code (see kerf_type)
        k -> i = value;
        return k;
}
```

7.31 `dotp` - Dot Product

`dotp(x, y)`

Calculate the dot product (or *scalar product*) of the vectors `x` and `y`. Equivalent to `sum x*y`.

```
KeRF> dotp(1 2 3, 1 2 5)
  20
```

7.32 `drop` - Drop Elements

`drop(x, y)`

Remove `x` elements from the beginning of the list `y`. Atomic types are unaffected by this operation.

```
KeRF> drop(3, "My Hero")
  "Hero"
KeRF> drop(5, 9 2 0)
  []
KeRF> drop(3, 5)
  5
```

The symbol `_` is equivalent to `drop` when used as a binary operator:

```
KeRF> 3 _ "My Hero"
  "Hero"
```

7.33 enlist - Enlist Element

enlist(x)

Wrap any element x in a list.

```
KeRF> enlist "A"
  ["A"]
KeRF> enlist 22 33
  [[22, 33]]
```

7.34 enum - Enumeration

enum(x)

Equivalent to hashed(x).

7.35 enumerate - Enumerate Items

enumerate(x)

If x is a number, generate a range of integers from 0 up to but not including x. Equivalent to til(x).

```
KeRF> enumerate 0
  INT[]
KeRF> enumerate 3
  [0, 1, 2]
KeRF> enumerate 5.3
  [0, 1, 2, 3, 4]
```

If x is a map, extract its keys.

```
KeRF> enumerate b:43, a:999
  ["b", "a"]
```

The symbol ^ is equivalent to enumerate when used as a unary operator:

```
KeRF> ^9
  [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

7.36 equal - Equal?

equal(x, y)

A predicate which returns 1 if x is equal to y. Equivalent to equals(x). Fully atomic.

```
KeRF> equal(5, 13)
  0
KeRF> equal(5, 5 13)
  [1, 0]
KeRF> equal(5 13, 5 13)
  [1, 1]
KeRF> equal(.1, .100000000000001)
  0
```

```
KeRF> equal(nan, nan)
  1
```

The symbols = and == are equivalent to `equal` when used as binary operators:

```
KeRF> 3 == 1 3 5
  [0, 1, 0]
```

7.37 `equals` - Equals?

`equals(x, y)`

A predicate which returns 1 if `x` is equal to `y`. Equivalent to `equal(x)`. Fully atomic.

7.38 `erf` - Error Function

`erf(x)`

Compute the Gauss error function of `x`. Atomic.

```
KeRF> erf -.5 -.2 0 .2 .3 1 2
  [-0.5205, -0.222703, 0, 0.222703, 0.328627, 0.842701, 0.995322]
```

7.39 `erfc` - Complementary Error Function

`erfc(x)`

Compute the complementary Gauss error function of `x`. Equivalent to `1 - erf(x)`. Atomic.

```
KeRF> erfc -.5 -.2 0 .2 .3 1 2
  [1.5205, 1.2227, 1, 0.777297, 0.671373, 0.157299, 0.00467773]
```

7.40 `eval` - Evaluate

`eval(x)`

Evaluate a string `x` as a KeRF expression. Partially atomic.

```
KeRF> a

KeRF> eval(["2+3", "a: 24", "a: 999"])
  [5, 24, 999]
KeRF> a
  999
```

7.41 `except` - Except

`except(x, y)`

Remove all the elements of `y` from `x`. Equivalent to `x[?!x in y]`.

```
KeRF> except("ABCDDBEFB", "ADF")
  "BCBEB"
```

If `x` is atomic, the result will be enclosed in a list:

```
KeRF> except(2, 3 4)
  [2]
```

7.42 exit - Exit

```
exit()
exit(x)
```

Exit the KeRF interpreter. If a number `x` is provided, use it as an exit code. Otherwise, exit with code 0 (successful).

```
KeRF> exit(1)
```

7.43 exp - Natural Exponential Function

```
exp(x)
exp(x, y)
```

Calculate e^x, the natural exponential function. If `y` is provided, calculate x^y. Fully atomic.

```
KeRF> exp 1 2 5
  [2.71828, 7.38906, 148.413]
KeRF> exp(2, 0 1 2)
  [1, 2, 4.0]
```

The symbol `**` is equivalent to `exp`:

```
KeRF> **1 2 5
  [2.71828, 7.38906, 148.413]
KeRF> 2**0 1 2
  [1, 2, 4.0]
```

7.44 explode - Explode

```
explode(x, y)
```

Violently and suddenly split the list `y` at instances of `x`. To reverse this process, use `implode`.

```
KeRF> explode(`"e", "A dream deferred")
  ["A dr", "am d", "f", "rr", "d"]
KeRF> explode(0, 1 1 2 0 2 0 5)
  [[1, 1, 2], [2], [5]]
```

Note that `explode` does not search for subsequences. Splitting on a 1-length string is not the same as splitting on a character:

```
KeRF> explode("rat", "drat, that rat went splat.")
  ["drat, that rat went splat."]
KeRF> explode("e", "A dream deferred")
  ["A dream deferred"]
```

7.45 first - First

`first(x)`
`first(x, y)`

When provided with a single argument, select the first element of the list `x`. Atomic types are unaffected by this operation.

```
KeRF> first(43 812 99 23)
  43
KeRF> first(99)
  99
```

When provided with two arguments, select the first `x` elements of `y`, repeating elements of `y` as necessary. Equivalent to `take(x, y)`.

```
KeRF> first(2, 43 812 99 23)
  [43, 812]
KeRF> first(8, 43 812 99 23)
  [43, 812, 99, 23, 43, 812, 99, 23]
```

7.46 flatten - Flatten

`flatten(x)`

Concatenate the elements of the list `x`. To join elements with a delimiter, use `implode`.

```
KeRF> flatten(["foo", "bar", "quux"])
  "foobarquux"
KeRF> flatten([2 3 4, 9 7 8, 14])
  [2, 3, 4, 9, 7, 8, 14]
```

Note that `flatten` only removes one level of nesting. To completely flatten an arbitrarily nested structure, combine it with `converge`:

```
KeRF> n: [[1,2],[3,4],[5,[6,7]]];
KeRF> flatten n
  [1, 2, 3, 4, 5, [6, 7]]
KeRF> flatten converge n
  [1, 2, 3, 4, 5, 6, 7]
```

7.47 float - Cast to Float

`float(x)`

Cast `x` to a float. Atomic.

```
KeRF> float 0 7 15
  [0, 7, 15.0]
```

When applied to a string, parse it into a number:

```
KeRF> float "97"
  97.0
```

7.48 `floor` - Floor

`floor(x)`

Compute the largest integer preceding a number `x`. Atomic.

```
KeRF> floor -3.2 0.4 0.9 1.1
  [-4, 0, 0, 1]
KeRF> int -3.2 0.4 0.9 1.1
  [-3, 0, 0, 1]
```

Taking the floor of a string or char converts it to lowercase:

```
KeRF> floor "Hello, World!"
  "hello, world!"
```

The symbol _ is equivalent to `floor` when used as a unary operator:

```
KeRF> _ 37.9 14.2
  [37, 14]
```

7.49 `greater` - Greater Than?

`greater(x, y)`

A predicate which returns 1 if `x` is greater than `y`. Fully atomic.

```
KeRF> greater(1 2 3, 2)
  [0, 0, 1]
KeRF> greater([5], [[], [3], [2 9]])
  [0, 1, 0]
KeRF> greater("apple", ["a", "aa", "banana"])
  [1, 1, 0]
```

The symbol > is equivalent to `greater` when used as a binary operator:

```
KeRF> 3 4 7 > 1 9 0
  [1, 0, 1]
```

7.50 `greatereq` - Greater or Equal?

`greatereq(x, y)`

A predicate which returns 1 if x is greater than or equal to y. Fully atomic.

```
KeRF> greatereq(1 2 3, 2)
  [0, 1, 1]
```

The symbol >= is equivalent to `greatereq` when used as a binary operator:

```
KeRF> 3 4 5 7 >= 1 9 5 0
  [1, 0, 1, 1]
```

7.51 `has_column` - Table Has Column?

`has_column(x, y)`

A predicate which returns 1 if table x has a column with the key y.

```
KeRF> has_column({{a: 1 2 3; b: 4 2 1}}, "a")
  1
KeRF> has_column({{a: 1 2 3; b: 4 2 1}}, "fictional")
  0
```

7.52 `has_key` - Has Key?

`has_key(x, y)`

A predicate which returns 1 if a map x contains the key y.

```
KeRF> m: {alphonse: 1, betty: 3, oscar: 99};
KeRF> has_key(m, "alphonse")
  1
KeRF> has_key(m, "alphys")
  0
```

If x is a list, return 1 if y is a valid index into x:

```
KeRF> l: 45 99 10 15;
KeRF> has_key(l, -1)
  0
KeRF> has_key(l, 2)
  1
KeRF> has_key(l, 2.2)
  1
KeRF> l[2.2]
  10
KeRF> l[-1]
  NAN
```

If x is a table, equivalent to `has_column(x, y)`.

7.53 `hash` - Hash

`hash(x)`

Equivalent to `hashed(x)`.

7.54 `hashed` - Hashed

`hashed(x)`

Create a list containing the elements of `x`, with hashmap-backed local interning. Interning will minimize the storage consumed by values which occur frequently and permit much more efficient sorting.

```
KeRF> data: rand(10000, ["apple", "pear", "banana"]);
KeRF> write_to_path("a.data", data);
KeRF> write_to_path("b.data", #data);
KeRF> system("wc -c a.data")
 1048576 a.data
KeRF> system("wc -c b.data")
  262144 b.data
```

The symbol `#` is equivalent to `hashed` when used as a unary operator:

```
KeRF> #["a", "b", "a"]
  #["a", "b", "a"]
```

7.55 `ident` - Identity

`ident(x)`

Unary identity function. Returns `x` unchanged.

```
KeRF> ident 42
  42
```

The symbol `:` is equivalent to `ident` when used as a unary operator:

```
KeRF> :42
  42
```

7.56 `ifnull` - If Null?

`ifnull(x)`

A predicate which returns 1 if `x` is null. Equivalent to `isnull(x)`. Atomic.

```
KeRF> ifnull([(), nan, 2, -3.7, [], {a:5}])
  [1, 1, 0, 0, [], {a:0}]
```

7.57 `implode` - Implode

`implode(x, y)`

Violently and suddenly join the elements of the list `y` intercalated with `x`. To reverse this process, use `explode`.

```
KeRF> implode("_and_", ["BIFF", "BOOM", "POW"])
  "BIFF_and_BOOM_and_POW"
KeRF> implode(23, 10 4 3 15)
  [10, 23, 4, 23, 3, 23, 15]
```

7.58 `in` - In?

`in(x, y)`

A predicate which returns 1 if each `x` is an element of `y`. Atomic over `x`.

```
KeRF> in(3, 8 7 3 2)
  [1]
KeRF> in(3 4, 8 7 3 2)
  [1, 0]
KeRF> in("a", "cassiopeia")
  [1]
```

7.59 `index` - Index

`index(x)`

Equivalent to `indexed(x)`.

7.60 `indexed` - Indexed

`indexed(x)`

Create an indexed version of a list `x`. This constructs an associated B-Tree, permitting faster searches and range queries. Do not use `indexed` if you know the list must always be ascending. The command `sort_debug` can be used to determine whether KeRF thinks a list is already sorted.

```
KeRF> indexed 3 7 0 5 2
  =[3, 7, 0, 5, 2]
```

The symbol = is equivalent to `indexed` when used as a unary operator:

```
KeRF> =3 2
  =[3, 2]
```

7.61 `int` - Cast to Int

`int(x)`

Cast `x` to an int, truncating. Atomic.

```
KeRF> int 33.6 -12.5 4 nan
  [33, -12, 4, NAN]
KeRF> floor 33.6 -12.5 4 nan
  [33, -13, 4, NAN]
```

When applied to a string, parse it into an integer:

```
KeRF> int ["1337", "47.2", ""]
  [1337, 47, 0]
```

7.62 `intersect` - Set Intersection

`intersect(x, y)`

Find unique items contained in both `x` and `y`. Equivalent to `distinct(x)[which distinct(x) in y]`.

```
KeRF> intersect(4, 4 5 6)
  [4]
KeRF> intersect(3 4, 1 2 3 4 5 6)
  [3, 4]
KeRF> intersect("ABD", "BCBD")
  "BD"
```

7.63 `isnull` - Is Null?

`isnull(x)`

A predicate which returns 1 if `x` is null. Atomic.

```
KeRF> isnull([(), nan, 2, -3.7, [], {a:5}])
  [1, 1, 0, 0, [], {a:0}]
```

7.64 `join` - Join

`join(x, y)`

Form a list by catenating `x` and `y`.

```
KeRF> join(2 3, 4 5)
  [2, 3, 4, 5]
KeRF> join(2 3, 4)
  [2, 3, 4]
KeRF> join(2 3, "ABC")
  [2, 3, `"A", `"B", `"C"]
KeRF> join({a:23, b:24}, {b:99})
  [{a:23, b:24}, {b:99}]
```

The symbol # is equivalent to `join` when used as a binary operator:

```
KeRF> 2 3 # 9
  [2, 3, 9]
```

7.65 json_from_kerf - Convert KeRF to JSON

json_from_kerf(x)

Convert a KeRF data structure x into a JSON (IETF RFC-4627) string.

```
KeRF> json_from_kerf({a: 45, b: [1, 3, 5.0]})
  "{\"a\":45,\"b\":[1,3,5]}"
KeRF> json_from_kerf({{a: 1 2 3, b: 4 5 6}})
  "{\"a\":[1,2,3],\"b\":[4,5,6],\"is_json_table\":[1]}"
```

7.66 kerf_from_json - Convert JSON to KeRF

kerf_from_json(x)

Convert a JSON (IETF RFC-4627) string x into a KeRF data structure. Note that booleans become the numbers 1 and 0 during this conversion process. KeRF-generated JSON strings generally contain the metadata necessary to round-trip without information loss, but JSON strings produced by another program may not.

```
KeRF> kerf_from_json("[23, 45, 9]")
  [23, 45, 9]
KeRF> kerf_from_json("[true, false]")
  [1, 0]
KeRF> kerf_from_json("{\"a\":[1,2,3],\"b\":[4,5,6]}")
  a:[1, 2, 3], b:[4, 5, 6]
KeRF> kerf_from_json("{\"a\":[1,2,3],\"b\":[4,5,6],\"is_json_table\":[1]}")
```

a	b
1	4
2	5
3	6

7.67 kerf_type - Type Code

kerf_type(x)

Obtain a numeric typecode from a KeRF value.

```
KeRF> kerf_type 45.0
  3
```

7.68 kerf_type_name - Type Name

kerf_type_name(x)

Obtain a human-readable type name string from a KeRF value. See kerf_type.

```
KeRF> kerf_type_name "Text"
  "character vector"
```

Type	Example	kerf_type_name	kerf_type
Timestamp Vector	[2000.01.01]	stamp vector	-4
Float Vector	[0.1]	float vector	-3
Integer Vector	[1]	integer vector	-2
Character Vector	"A"	character vector	-1
Function	{[x] 1+x}	function	0
Character	`A	character	1
Integer	1	integer	2
Float	0.1	float	3
Timestamp	2000.01.01	stamp	4
Null	()	null	5
List	[]	list	6
Map	{a:1}	map	7
Enumeration	enum ["a"]	enum	8
Index	index [1,2]	sort	9
Table	{{a:1}}	table	10
Atlas	atlas {a:1}	atlas	11

KeRF types

7.69 last - Last

```
last(x)
last(x, y)
```

When provided with a single argument, select the last element of the list x. Atomic types are unaffected by this operation.

```
KeRF> last(43 812 99 23)
  23
KeRF> last(99)
  99
```

When provided with two arguments, select the last x elements of y, repeating elements of y as necessary. Equivalent to take(-x, y).

```
KeRF> last(2, 43 812 99 23)
  [99, 23]
KeRF> last(7, 43 812 99 23)
  [812, 99, 23, 43, 812, 99, 23]
```

7.70 `left_join` - Left Join

`left_join(x, y, z)`

Peform a left join of the tables x and y on the column z. A left join includes every row of the left table (x), and adds any additional columns from the right table (y) by matching on some key column (y). Added columns where there is no match on z will be filled with type-appropriate null values as generated by `type_null`.

```
KeRF> t: {{a:1 2 2 3, b:10 20 30 40}}
```

a	b
1	10
2	20
2	30
3	40

```
KeRF> u: {{a:2 3, c:1.5 3}}
```

a	c
2	1.5
3	3.0

```
KeRF> left_join(t, u, "a")
```

a	b	c
1	10	nan
2	20	1.5
2	30	1.5
3	40	3.0

If z is a list, require a match on several columns:

```
KeRF> u: {{a:2 3, b:30 40, c:1.5 3}};
KeRF> left_join(t, u, ["a","b"])
```

a	b	c
1	10	nan
2	20	nan
2	30	1.5
3	40	3.0

If z is a map, associate columns from x as keys with columns from y as values, permitting joins across tables whose column names differ.

```
KeRF> u: {{z:2 3, c:1.5 3}};
KeRF> left_join(t, u, {'a':'z'})
```

a	b	c
1	10	nan
2	20	1.5
2	30	1.5
3	40	3.0

7.71 `len` - Length

`len(x)`

Determine the number of elements in x. Equivalent to `count(x)`. Atomic elements have a count of 1.

```
KeRF> len 4 7 9
  3
KeRF> len [4 7 9, 23 32]
  2
KeRF> len 5
  1
KeRF> len {a:23, b:45}
  1
```

7.72 `less` - Less Than?

`less(x, y)`

A predicate which returns 1 if x is less than y. Fully atomic.

```
KeRF> less(1 2 3, 2)
  [1, 0, 0]
KeRF> less([5], [[], [3], [2 9]])
  [1, 0, 1]
KeRF> less("apple", ["a", "aa", "banana"])
  [0, 0, 1]
```

The symbol < is equivalent to `less` when used as a binary operator:

```
KeRF> 3 4 7 < 1 9 0
  [0, 1, 0]
```

7.73 `lesseq` - Less or Equal?

`lesseq(x, y)`

A predicate which returns 1 if x is less than or equal to x. Fully atomic.

```
KeRF> lesseq(1 2 3, 2)
  [1, 1, 0]
KeRF> lesseq([5], [[], [3], [5], [2 9]])
  [1, 0, 0, 1]
KeRF> lesseq("apple", ["a", "aa", "apple", "banana"])
  [0, 0, 1, 1]
```

The symbol <= is equivalent to `lesseq` when used as a binary operator:

```
KeRF> 3 4 1 7 <= 1 9 1 0
  [0, 1, 1, 0]
```

7.74 `lg` - Base 2 Logarithm

`lg(x)`

Calculate $\log_2(x)$. Equivalent to `log(2, x)`. Atomic.

```
KeRF> lg 128 512 37
  [7, 9, 5.20945]
```

7.75 `lines` - Lines From File

`lines(filename)`
`lines(filename, n)`

Load lines from `filename` into a list of strings. If `n` is present, limit loading to `n` lines.

```
KeRF> lines("example.txt")
  ["First line", "Second line", "Third line"]
KeRF> lines("example.txt", 2)
  ["First line", "Second line"]
```

7.76 `ln` - Natural Logarithm

`ln(x)`

Calculate $\log_e(x)$. Atomic.

```
KeRF> ln 2 3 10 37
  [0.693147, 1.09861, 2.30259, 3.61092]
```

7.77 `load` - Load Source

`load(filename)`

Load and run KeRF source from a file. Given an example file:

```
// comment
a: 7+range 10
b: range 10
a * b
```

Loading the file from the Repl:

```
KeRF> load("manual/example.kerf")
KeRF> a
  [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
KeRF> b
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that the value of the raw expression `a * b` is not printed when you `load` a file. If this is desired, use `display` in the script.

7.78 `log` - Logarithm

```
log(x)
log(x, y)
```

Calculate the base `x` logarithm of `y`. If only one argument is provided, `log(10, x)` is assumed. Fully atomic.

```
KeRF> log 3 8 10 16 100
  [0.477121, 0.90309, 1, 1.20412, 2.0]
KeRF> log(2, 3 8 10 16 100)
  [1.58496, 3, 3.32193, 4, 6.64386]
KeRF> log(2 3 4, 8)
  [3, 1.89279, 1.5]
```

7.79 `lsq` - Least Squares Solution

```
lsq(A, B)
```

Solve $Ax = B$ for x, where `A` is a matrix and `B` is a matrix or vector.

```
KeRF> lsq([1 2;4 4], [3 4])
  [[1, 0.5]]
KeRF> lsq([0 .5;2 0], [0 1;1 0])
  [[2, 0.0],
   [0, 0.5]]
```

The symbol \ is equivalent to `lsq` when used as a binary operator:

```
KeRF> [0 .5;2 0]\[0 1;1 0]
  [[2, 0.0],
   [0, 0.5]]
```

7.80 `map` - Make Map

```
map(x, y)
```

Make a map from a list of keys `x` and a list of values `y`. These lists must be the same length.

```
KeRF> map("ABC", 44 18 790)
  {`"A":44, `"B":18, `"C":790}
```

The symbol `!` is equivalent to `map` when used as a binary operator:

```
KeRF> "ABC" ! 44 18 790
  {`"A":44, `"B":18, `"C":790}
```

7.81 match - Match?

`match(x, y)`

A predicate which returns 1 if `x` is identical to `y`. While `equals` compares atoms, `match` is not atomic and compares entire values.

```
KeRF> match(5, 3 5 7)
  0
KeRF> match(3 5 7, 3 5 7)
  1
```

The symbol `~` is equivalent to `match` when used as a binary operator:

```
KeRF> 3 5 7 ~ 3 5 7
  1
```

7.82 mavg - Moving Average

`mavg(x, y)`

For a series of sliding windows of size `x` ending at each element of the list `y`, find the arithmetic mean of valid (not nan and in range of the list) elements. Equivalent to `msum(x, y)/mcount(x, y)`.

```
KeRF> mavg(3, 1 2 4 2 1 nan 0 4 6 7)
  [1, 1.5, 2.33333, 2.66667, 2.33333, 1.5, 0.5, 2, 3.33333, 5.66667]
```

7.83 max - Maximum

`max(x)`

Find the maximum element of `x`. Roughly equivalent to `last sort`, but much more efficient.

```
KeRF> max 7 0 15 -1 8
  15
KeRF> max 0 -inf -5 nan inf
  inf
```

7.84 maxes - Maximums

`maxes(x, y)`

Find the maximum of `x` and `y`. Fully atomic.

```
KeRF> maxes(1 3 7 2 0 1, -4 3 20 1 9 4)
  [1, 3, 20, 2, 9, 4]
KeRF> maxes(8, -4 3 20 1 9 4)
  [8, 8, 20, 8, 9, 8]
```

The symbol | is equivalent to maxes when used as a binary operator:

```
KeRF> -4 3 20 1 9 4 | 15
  [15, 15, 20, 15, 15, 15]
```

7.85 mcount - Moving Count

mcount(x, y)

For a series of sliding windows of size x ending at each element of the list y, count the number of valid (not nan and in range of the list) elements. Equivalent to msum(x, not isnull y).

```
KeRF> mcount(3, 1 2 3 nan 4 5 nan nan nan)
  [1, 2, 3, 2, 2, 2, 2, 1, 0]
```

7.86 meta_table - Meta Table

meta_table(x)

Produce a table containing debugging metadata about some some table x:

```
KeRF> meta_table {{a: 1 2 3, b: 3.0 17 4}}
```

column	type	type_name	is_ascending	is_disk
a	-2	integer vector	1	0
b	-3	float vector	0	0

7.87 min - Minimum

min(x)

Find the minimum element of x. Roughly equivalent to first sort, but much more efficient.

```
KeRF> min 7 0 15 -1 8
  -1
KeRF> min 0 -inf -5 nan inf
  -inf
```

7.88 mins - Minimums

mins(x, y)

Find the minimum of x and y. Fully atomic.

```
KeRF> mins(1 3 7 2 0 1, -4 3 20 1 9 4)
  [-4, 3, 7, 1, 0, 1]
KeRF> mins(8, -4 3 20 1 9 4)
  [-4, 3, 8, 1, 8, 4]
```

The symbol & is equivalent to mins when used as a binary operator:

```
KeRF> -4 3 20 1 9 4 & 15
  [-4, 3, 15, 1, 9, 4]
```

7.89 `minus` - Minus

`minus(x, y)`

Calculate the difference of `x` and `y`. Fully atomic.

```
KeRF> minus(3, 5)
  -2
KeRF> minus(3, 9 15 -7)
  [-6, -12, 10]
KeRF> minus(9 15 -7, 3)
  [6, 12, -10]
KeRF> minus(9 15 -7, 1 3 5)
  [8, 12, -12]
```

The symbol `-` is equivalent to `minus` when used as a binary operator:

```
KeRF> 2 4 3 - 9
  [-7, -5, -6]
```

7.90 `minv` - Matrix Inverse

`minv(x)`

Calculate the inverse of a matrix `x`.

```
KeRF> minv([1 2;3 4])
  [[-2, 1],
 [1.5, -0.5]]
```

7.91 `mmax` - Moving Maximum

`mmax(x, y)`

For a series of sliding windows of size `x` ending at each element of the list `y`, find the maximum element. Equivalent to `(x-1) or mapback converge y`.

```
KeRF> mmax(3, 0 1 0 2 0 1 0)
  [0, 1, 1, 2, 2, 2, 1]
```

7.92 `mmin` - Moving Minimum

`mmin(x, y)`

For a series of sliding windows of size `x` ending at each element of the list `y`, find the minimum element. Equivalent to `(x-1) and mapback converge y`.

```
KeRF> mmin(3, 4 0 3 0 2 0 4 5 6)
  [4, 0, 0, 0, 0, 0, 0, 0, 4]
```

7.93 `mmul` - Matrix Multiply

`mmul(x, y)`

Multiply the matrix or vector `x` by the matrix or vector `y`. Equivalent to `x dotp mapleft y`.

```
KeRF> mmul([1 2;3 4], [5 6])
  [[15, 18],
   [35, 42]]
KeRF> mmul([1 2;3 4], [0 1;1 0])
  [[2, 1],
   [4, 3]]
```

7.94 `mod` - Modulus

`mod(x, y)`

Calculate `x` modulo `y`. Equivalent to `x - y * floor(x/y)`. Left-atomic.

```
KeRF> mod(0 1 2 3 4 5 6 7, 3)
  [0, 1, 2, 0, 1, 2, 0, 1]
KeRF> mod(-4 -3 -2 -1 0 1 2, 2)
  [0, 1, 0, 1, 0, 1, 0]
```

The symbol % is equivalent to `mod` when used as a binary operator:

```
KeRF> 0 1 2 3 4 5 6 7 % 3
  [0, 1, 2, 0, 1, 2, 0, 1]
```

7.95 `msum` - Moving Sum

`msum(x, y)`

Calculate a series of sums of each element in a list `y` and up to the `x` previous values, ignoring nans and nonexistent values.

```
KeRF> msum(2, 10 20 30 40)
  [10, 30, 50, 70]
KeRF> msum(2, 1 2 2 nan 1 2)
  [1, 3, 4, 2, 1, 3.0]
KeRF> msum(3, 10 10 14 10 25 10 Nan 10)
  [10, 20, 34, 34, 49, 45, 35, 20.0]
```

`msum(1, y)` can be used to remove nan from data:

```
KeRF> msum(1, 4 2 1 nan 2)
  [4, 2, 1, 0, 2.0]
```

7.96 `negate` - Negate

`negate(x)`

Reverse the sign of a number `x`. Equivalent to `-1 * x`. Atomic.

```
KeRF> negate 2 4 -77
  [-2, -4, 77]
```

The symbol - is equivalent to negate when used as a binary operator:

```
KeRF> -(2 4 -77)
  [-2, -4, 77]
```

7.97 negative - Negative

negative(x)

Equivalent to negate(x).

7.98 not - Logical Not

not(x)

Calculate the logical *NOT* of x. Atomic.

```
KeRF> not(1 0)
  [0, 1]
KeRF> not([0, -4, 9, nan, []])
  [1, 0, 0, 0, []]
```

The symbol ! is equivalent to not when used as a unary operator:

```
KeRF> !1 0 8
  [0, 1, 0]
```

7.99 noteq - Not Equal?

noteq(x, y)

A predicate which returns 1 if x is not equal to y. Equivalent to not equals(x). Fully atomic.

```
KeRF> noteq(5, 13)
  1
KeRF> noteq(5, 5 13)
  [0, 1]
KeRF> noteq(5 13, 5 13)
  [0, 0]
KeRF> noteq(.1, .100000000000001)
  1
KeRF> noteq(nan, nan)
  0
```

The symbols != and <> are equivalent to noteq when used as binary operators:

```
KeRF> 3 != 1 3 5
  [1, 0, 1]
```

7.100 now - Current DateTime

now()

Return a stamp containing the current date and time in UTC.

```
KeRF> now()
  2015.10.31T21:14:09.018
```

7.101 now_date - Current Date

now_date()

Return a stamp containing the current date only in UTC.

```
KeRF> now_date()
  2015.10.31
```

7.102 now_time - Current Time

now_time()

Return a stamp containing the current time only in UTC.

```
KeRF> now_time()
  21:14:09.018
```

7.103 open_socket - Open Socket

open_socket(host, port)

Establish a connection to a remote KeRF instance at hostname host and listening on port (This port is specified via the -p command-line argument when starting a KeRF process) and return a connection handle. Both host and port must be strings. Once opened, this handle may be used via send_async or send_sync. When a connection is no longer needed, it can be closed via close_socket.

```
KeRF> open_socket("localhost", "1234")
  4
```

7.104 open_table - Open Table

open_table(filename)

Load a serialized table from the binary file filename. Tables can be explicitly serialized via write_to_path.

```
KeRF> write_to_path("temp.table", {{a:1 2 3}})
  0
KeRF> exit()
> ./kerf -q
KeRF> open_table("temp.table")

┌─┐
│a│
```

```
1
2
3
```

7.105 or - Logical OR

`or(x, y)`

Calculate the logical *OR* of x and y. This operation is equivalent to the primitive function max. Fully atomic.

```
KeRF> or(1 1 0 0, 1 0 1 0)
  [1, 1, 1, 0]
KeRF> or(1 2 3 4, 0 -4 9 0)
  [1, 2, 9, 4]
```

The symbol | is equivalent to or when used as a binary operator:

```
KeRF> 1 1 0 0 | 1 0 1 0
  [1, 1, 1, 0]
```

7.106 order - Order

`order(x)`

Generate a list of indices showing the relative ascending order of items in the list x. Equivalent to <<x.

```
KeRF> order "ABCEDF"
  [0, 1, 2, 4, 3, 5]
KeRF> order 2 4 1 9
  [1, 2, 0, 3]
KeRF> <2 4 1 9
  [2, 0, 1, 3]
KeRF> <<2 4 1 9
  [1, 2, 0, 3]
```

7.107 out - Output

`out(x)`

Print a string x to stdout. Non-string values are ignored.

```
KeRF> out "foo"
foo
KeRF> out 65
KeRF>
```

7.108 part - Partition

`part(x)`

Produce a map from unique elements of a list x to lists of the indices at which these elements could originally be found.

```
KeRF> part 3 5 7 7 5
  {3:[0], 5:[1, 4], 7:[2, 3]}
KeRF> part ["apple", "frog", "frog", "kumquat"]
  {apple:[0], frog:[1, 2], kumquat:[3]}
```

`part` does not affect atomic types:

```
KeRF> part {a: 23 45, b: 9}
  {a:[23, 45], b:9}
KeRF> part 23
  23
```

The symbol `&` is equivalent to `part` when used as a unary operator:

```
KeRF> &2 2 1 2
  {2:[0, 1, 3], 1:[2]}
```

7.109 `plus` - Plus

`plus(x, y)`

Equivalent to `add`.

7.110 `pow` - Exponentiation

`pow(x)`
`pow(x, y)`

Equivalent to `exp`.

7.111 `rand` - Random Numbers

`rand()`
`rand(x)`
`rand(x, y)`

Generate a random integer vector of `x` integers from 0 up to but not including `y`. If `x` is negative, draw numbers in the given range without replacement. That is, drawn numbers will be unique.

```
KeRF> rand(10, 3)
  [0, 2, 1, 2, 1, 2, 1, 2, 0, 2]
KeRF> rand(-5, 10)
  [3, 9, 4, 0, 8]
```

If `y` is a list, select random elements from `y`. As above, a negative `x` produces draws without replacement:

```
KeRF> rand(6, "ABC")
  "CBCBBA"
KeRF> rand(-7, "ABCDEFG")
  "DAGBFCE"
```

If y is not provided, generate a single random integer from 0 up to but not including x. As above, if x is a list, draw a single random element.

```
KeRF> rand(10)
  1
KeRF> rand(10)
  8
KeRF> rand(10)
  7
KeRF> rand("ABCDE")
  `"B"
```

If rand is given no arguments, generate a single random float from 0 up to but not including 1.

```
KeRF> rand()
  0.389022
```

The symbol ? is equivalent to rand when used as a binary operator:

```
KeRF> 5?2
  [0, 0, 1, 0, 0]
```

7.112 range - Range

range(x)
range(x, y)
range(x, y, z)

If range is provided with one argument, generate a vector of integers from 0 up to but not including x:

```
KeRF> range 5
  [0, 1, 2, 3, 4]
```

If range is provided with two arguments, generate a vector of numbers from x up to but not including y, spaced 1 apart:

```
KeRF> range(10, 15)
  [10, 11, 12, 13, 14]
KeRF> range(10.5, 16.5)
  [10.5, 11.5, 12.5, 13.5, 14.5, 15.5]
```

If range is provided with three arguments, generate a vector of numbers from x up to but not including y, spaced z apart:

```
KeRF> range(1, 3, .3)
  [1, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8]
```

7.113 read_from_path - Read From Path

read_from_path(filename)

Load a serialized KeRF data structure from the binary file filename.

```
KeRF> write_to_path("temp.dat", [2, 7, 15])
  0
KeRF> read_from_path("temp.dat")
  [2, 7, 15]
```

7.114 read_table_from_csv - Read Table From CSV File

`read_table_from_csv(filename, fields, n)`

Load a Comma-Separated Value file into a table. `fields` is a string which indicates the expected datatype of each column in the CSV file- see `read_table_from_delimited_file` for the supported column types and their symbols. `n` indicates how many rows of the file are treated as column headers- generally 0 or 1.

Equivalent to `read_table_from_delimited_file(",", filename, fields, n)`.

7.115 read_table_from_delimited_file - Read Table From Delimited File

`read_table_from_delimited_file(delimiter, filename, fields, n)`

Load the contents of a text file with rows separated by newlines and fields separated by some character `delimiter` into a table. `fields` is a string which indicates the expected datatype of each column. `n` indicates how many rows of the file are treated as column headers- generally 0 or 1.

Symbol	Datatype
`I`	Integer
`F`	Float
`S`	String
`E`	Enumerated String
`G`	IETF RFC-4122 UUID
`N`	IP address as parsed by `inet_pton()`
`Z`	Custom Datetime. See `.Parse.strptime_format`
`*`	Skipped field

Symbols accepted as part of `fields`

Given a file like the following:

```
Language&Lines&Runtime
C&271&0.101
Java&89&0.34
Python&62&3.79
```

```
KeRF> read_table_from_delimited_file("&", "code.txt", "SIF", 1)

┌────────┬─────┬───────┐
│Language│Lines│Runtime│
├────────┼─────┼───────┤
│       C│  271│  0.101│
│    Java│   89│   0.34│
│  Python│   62│   3.79│
└────────┴─────┴───────┘
```

7.116 `read_table_from_fixed_file` - Read Table From Fixed-Width File

`read_table_from_fixed_file(filename, attributes)`
Load the contents of a text file with fixed-width columns. The map `attributes` specifies the details of the format:

Key	Type	Optional	Description
`fields`	String	No	As in `read_table_from_delimited_file`.
`widths`	Integer Vector	No	The width of each column in characters.
`line_limit`	Integer	Yes	The maximum number of rows to load.
`titles`	List of String	Yes	Key for each column in the resulting table.
`header_rows`	Integer	Yes	How many rows are treated as column headers.
`newline_separated`	Boolean	Yes	if false, do not expect newlines separating rows.

Settings described in `attributes`

Symbol	Datatype
`Y`	NYSE TAQ symbol
`Q`	NYSE TAQ time format (`HHMMSSXXX`)

Additional symbols accepted as part of `fields` in fixed-width files

Given a file like this list of ingredients for my *famous* pizza dough:

```
Eggs      2.0 -
Flour     5.0 cups
Honey     2.0 tbs
Water     2.0 cups
Olive Oil 2.0 tbsp
Yeast     1.0 tbsp
```

```
KeRF> fmt: { fields: "SFS", widths: [10, 4, 5], titles: ["Ingredient", "Amount", "Unit"] };
KeRF> read_table_from_fixed_file("dough.txt", fmt)
```

Ingredient	Amount	Unit
Eggs	2.0	-
Flour	5.0	cups
Honey	2.0	tbsp
Water	2.0	cups
Olive Oil	2.0	tbsp
Yeast	1.0	tbsp

7.117 `read_table_from_tsv` - Read Table From TSV File

`read_table_from_tsv(filename, fields, n)`

Load a Tab-Separated Value file into a table. `fields` is a string which indicates the expected datatype of each column in the TSV file- see `read_table_from_delimited_file` for the supported column types and their symbols. `n` indicates how many rows of the file are treated as column headers- generally 0 or 1.

Equivalent to `read_table_from_delimited_file("\t", filename, fields, n)`.

7.118 rep - Output Representation

```
rep(x)
```

Convert a value `x` into a printable string representation. If you only wish to convert the atoms of `x` into strings, use `string`.

```
KeRF> rep 45
  "45"
KeRF> rep 2 5 3
  "[2, 5, 3]"
KeRF> rep {a:4}
  "{a:4}"
KeRF> rep "Some text"
  "\"Some text\""
```

7.119 repeat - Repeat

```
repeat(x, y)
```

Create a list containing `x` copies of `y`. Equivalent to `x take enlist y`.

```
KeRF> repeat(2, 5)
  [5, 5]
KeRF> repeat(4, "AB")
  ["AB", "AB", "AB", "AB"]
KeRF> repeat(0, "AB")
  []
KeRF> repeat(-3, "AB")
  []
```

7.120 reserved - Reserved Names

```
reserved()
```

Print and return an unsorted list of KeRF's reserved names, including reserved literals such as `true`.

7.121 reverse - Reverse

```
reverse(x)
```

Reverse the order of the elements of the list `x`. Atomic types are unaffected by this operation.

```
KeRF> reverse 23 78 94
  [94, 78, 23]
KeRF> reverse "backwards"
  "sdrawkcab"
KeRF> reverse 5
  5
KeRF> reverse {a: 23 56, b:0 1}
  a:[23, 56], b:[0, 1]
```

The symbol / is equivalent to `reverse` when used as a unary operator:

```
KeRF> /"example text"
  "txet elpmaxe"
```

7.122 rsum - Running Sum

`rsum(x)`

Calculate a running sum of the elements of the list `x`, from left to right. `nan`s are ignored.

```
KeRF> rsum 1
  1
KeRF> rsum 1 2
  [1, 3]
KeRF> rsum 1 2 5
  [1, 3, 8]
KeRF> rsum 1 2 5 7 8
  [1, 3, 8, 15, 23]
KeRF> rsum 1 2 3 nan 4
  [1, 3, 6, 6, 10.0]
KeRF> rsum []
  []
```

7.123 run - Run

`run(filename)`

Load and run KeRF source from a file. Equivalent to `load(filename)`.

7.124 send_async - Send Asynchronous

`send_async(x, y)`

Given a connection handle `x`, as obtained with `open_socket`, send a string `y` to a remote KeRF instance and do not wait for a reply. `y` will be `eval`ed on the remote server. Returns 1 on a successful send.

```
KeRF> c: open_socket("localhost", "1234")
  4
KeRF> send_async(c, "foo: 2+3")
  1
KeRF> foo
  {}
KeRF> send_sync(c, "[foo, foo]")
  [5, 5]
```

7.125 send_sync - Send Synchronous

`send_sync(x, y)`

Given a connection handle `x`, as obtained with `open_socket`, send a string `y` to a remote KeRF instance, waiting for a reply. `y` will be `eval`ed on the remote server, and the result will be returned.

```
KeRF> c: open_socket("localhost", "1234")
  4
KeRF> send_sync(c, "sum range 1000")
  499500
```

7.126 setminus - Set Disjunction

`setminus(x, y)`

Equivalent to `except(x, y)`.

7.127 shift - Shift

`shift(x, y)`
`shift(x, y, z)`

Offset `y` by `x` positions, filling shifted-in positions with `z`.

```
KeRF> shift(4, 1 2 3 4 5 6 7, 999)
  [999, 999, 999, 999, 1, 2, 3]
KeRF> shift(-1, 1 2 3 4 5 6 7, 999)
  [2, 3, 4, 5, 6, 7, 999]
```

If `z` is not provided, use a type-appropriate null value as generated by `type_null`.

```
KeRF> shift(3, "ABCDE")
  "   AB"
KeRF> shift(-3, "ABCDE")
  "DE   "
KeRF> shift(2, 1 2 3 4)
  [NAN, NAN, 1, 2]
KeRF> shift(2, 1.0 2.0 3.0 4.0)
  [nan, nan, 1, 2.0]
```

7.128 shuffle - Shuffle

`shuffle(x)`

Randomly permute the elements of the list `x`. Equivalent to `rand(-len(x), x)`.

```
KeRF> shuffle "APPLE"
  "PAEPL"
KeRF> shuffle "APPLE"
  "LEAPP"
KeRF> shuffle "APPLE"
  "LEPAP"
```

7.129 sin - Sine

`sin(x)`

Calculate the sine of `x`, expressed in radians. Atomic. The results of `sin` will always be floating point values.

```
KeRF> sin 3.14159 1 -20
  [2.65359e-06, 0.841471, -0.912945]
KeRF> asin sin 3.14159 1 -20
  [2.65359e-06, 1, -1.15044]
```

7.130 sinh - Hyperbolic Sine

`sinh(x)`

Calculate the hyperbolic sine of `x`, expressed in radians. Atomic. The results of `sinh` will always be floating point values.

```
KeRF> sinh 3.14159 1 -20
  [11.5487, 1.1752, -2.42583e+08]
```

7.131 sleep - Sleep

`sleep(x)`

Delay for at least `x` milliseconds and then return `x`.

```
> time ./kerf -x "sleep 3000"
3000

real    0m3.070s
user    0m0.002s
sys     0m0.063s
```

7.132 sort - Sort

`sort(x)`

Sort the elements of the list `x` in ascending order. Equivalent to `x[ascend x]`.

```
KeRF> sort "ALPHABETICAL"
  "AAABCEHILLPT"
KeRF> sort 27 18 4 9
  [4, 9, 18, 27]
KeRF> sort unique "how razorback jumping frogs can level six piqued gymnasts"
  " abcdefghijklmnopqrstuvwxyz"
```

7.133 sort_debug - Sort Debug

`sort_debug(x)`

Print debugging information about the list `x` which is relevant to the performance and internal datapaths of searching and sorting and then return `x`. The precise format in which this function prints its results is likely to change in the future.

```
KeRF> sort_debug range(4);
is array: 1
```

```
ATTR_SORTED: 1
Actually sorted array: 1
Mismatch: 0
KeRF> sort_debug "ACED";
is array: 1
ATTR_SORTED: 0
Actually sorted array: 0
Mismatch: 0
KeRF> sort_debug 27;
is array: 0
ATTR_SORTED: 1
Actually sorted array: 1
Mismatch: 0
```

7.134 sqrt - Square Root

sqrt(x)

Calculate the square root of x. Atomic.

```
KeRF> sqrt 2 25 100
  [1.41421, 5, 10.0]
```

7.135 stamp_diff - Timestamp Difference

stamp_diff(x, y)

Calculate the difference between the timestamps x and y in nanoseconds.

```
KeRF> t: now(); sleep(10); stamp_diff(now(), t)
  10302000
KeRF> t: now(); sleep(10); stamp_diff(t, now())
  -10874000
```

7.136 std - Standard Deviation

std(x)

Calculate the standard deviation of the elements of the list x. Equivalent to sqrt var x.

```
KeRF> std 4 7 19 2 0 -2
  6.87992
KeRF> std {a: 4 1 0}
  {a:1.69967}
```

7.137 string - Cast to String

string(x)

Convert the value x to a string. Atomic. If you wish to recursively convert an entire data structure to a string, use rep.

```
KeRF> string 990
  "990"
KeRF> string 15 9 10
  ["15", "9", "10"]
KeRF> string {a:4 5}
  {a:["4", "5"]}
```

7.138 subtract - Subtract

`subtract(x, y)`

Calculate the difference of x and y. Fully atomic. Equivalent to `minus(x, y)`.

7.139 sum - Sum

`sum(x)`

Calculate the sum of the elements of the list x.

```
KeRF> sum 4 3 9
  16
KeRF> sum 5
  5
KeRF> sum []
  0
```

7.140 system - System

`system(x)`

Execute a string x containing a system command as if from `/bin/sh -c x`.

```
KeRF> system("cal")
   November 2015
Su Mo Tu We Th Fr Sa
 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
KeRF> system("echo \"hello\"")
hello
```

7.141 tables - Tables

`tables()`

Generate a list of the names of all currently loaded tables.

```
KeRF> tables()
  []
KeRF> t: {{a: 1 2 3, b: 4 5 6}};
```

```
KeRF> tables()
  ["t"]
```

7.142 take - Take

`take(x, y)`

Create a list containing the first x elements of y, looping y as necessary. If x is negative, take backwards from the last to the first. Equivalent to `first(x, y)`.

```
KeRF> take(3, `"A")
  "AAA"
KeRF> take(2, range(5))
  [0, 1]
KeRF> take(8, range(5))
  [0, 1, 2, 3, 4, 0, 1, 2]
KeRF> take(-3, range(5))
  [2, 3, 4]
```

The symbol ^ is equivalent to `take` when used as a binary operator:

```
KeRF> 3^"ABCDE"
  "ABC
```

7.143 tan - Tangent

`tan(x)`

Calculate the tangent of x, expressed in radians. Atomic. The results of `tan` will always be floating point values.

```
KeRF> tan 0.5 -0.2 1 4
  [0.546302, -0.20271, 1.55741, 1.15782]
KeRF> atan(tan 0.5 -0.2 1 4)
  [0.5, -0.2, 1, 0.858407]
```

7.144 tanh - Hyperbolic Tangent

`tanh(x)`

Calculate the hyperbolic tangent of x, expressed in radians. Atomic. The results of `tanh` will always be floating point values.

```
KeRF> tanh 3.14159 1 -20
  [0.996272, 0.761594, -1.0]
```

7.145 times - Multiplication

`times(x, y)`

Calculate the product of x and y. Fully atomic.

```
KeRF> times(3, 5)
  15
KeRF> times(1 2 3, 5)
  [5, 10, 15]
KeRF> times(3, 5 8 9)
  [15, 24, 27]
KeRF> times(10 15 3, 8 2 4)
  [80, 30, 12]
```

The symbol * is equivalent to `times` when used as a binary operator:

```
KeRF> 1 2 3*2
  [2, 4, 6]
```

7.146 `timing` - Timing

`timing(x)`

If `x` is truthy, enable timing. Otherwise, disable it. Returns a boolean timing status. When timing is active, all operations will print their approximate runtime in milliseconds after completing.

```
KeRF> timing(1)
  1
KeRF> sum range exp(2, 24)
  140737479966720

    203 ms
KeRF> timing(0)
  0
```

7.147 `tolower` - To Lowercase

`tolower(x)`

Convert a string `x` to lowercase. Equivalent to `floor(x)`.

7.148 `toupper` - To Uppercase

`toupper(x)`

Convert a string `x` to uppercase. Equivalent to `ceil(x)`.

7.149 `transpose` - Transpose

`transpose(x)`

Take the transpose (flip the x and y axes) of a matrix `x`. Has no effect on atoms or lists of atoms.

```
KeRF> transpose [1 2 3, 4 5 6, 7 8 9]
  [[1, 4, 7],
   [2, 5, 8],
   [3, 6, 9]]
```

```
KeRF> transpose 1 2 3
  [1, 2, 3]
```

Atoms will "spread" as needed to produce a rectangular matrix if the list contains any sublists:

```
KeRF> transpose [2, 3 4 5, 6]
  [[2, 3, 6],
   [2, 4, 6],
   [2, 5, 6]]
```

The symbol + is equivalent to `transpose` when used as a unary operator:

```
KeRF> +[1 2, 3 4]
  [[1, 3],
   [2, 4]]
```

7.150 trim - Trim

`trim(x)`

Remove leading and trailing whitespace from strings. Atomic.

```
KeRF> trim(" some text   ")
  "some text"
KeRF> trim [" some text   ", "    another", "\tab\t"]
  ["some text", "another", "ab"]
```

7.151 type_null - Type Null

`type_null(x)`

Generate the equivalent type-specific null value for x.

```
KeRF> type_null("ABC")
  `" "
KeRF> type_null(1.0)
  nan
KeRF> type_null(1)
  NAN
KeRF> type_null(now())
  00:00:00.000
```

7.152 uneval - Uneval

`uneval(x)`

Equivalent to `json_from_kerf(x)`.

7.153 union - Set Union

`union(x, y)`

Construct an unsorted list of the unique elements in either x or y. Equivalent to `distinct join(x, y)`.

```
KeRF> union(2, 4 5)
  [2, 4, 5]
KeRF> union([], 2)
  [2]
KeRF> union(2 3 4, 1 3 9)
  [2, 3, 4, 1, 9]
```

7.154 unique - Unique Elements

`unique(x)`

Equivalent to `distinct(x)`.

7.155 var - Variance

`var(x)`

Calculate the variance of the elements of a list `x`. Equivalent to `(sum (x - avg x)**2)/count_nonnull x`.

```
KeRF> var []
  nan
KeRF> var 4 3 8 2
  5.1875
KeRF> sqrt var 4 3 8 2
  2.27761
```

7.156 which - Which

`which(x)`

For each index i of the list `x`, produce `x[i]` copies of `i`:

```
KeRF> which 1 2 1 4
  [0, 1, 1, 2, 3, 3, 3, 3]
KeRF> which 1 2 3
  [0, 1, 1, 2, 2, 2]
KeRF> which 1 1 1
  [0, 1, 2]
```

This operation is most often used to retrieve a list of the indices of nonzero elements of a boolean vector:

```
KeRF> which 0 0 1 0 1 1 0 1
  [2, 4, 5, 7]
```

The symbol `?` is equivalent to `which` when used as a binary operator:

```
KeRF> ?0 0 1 0 1 1 0 1
  [2, 4, 5, 7]
```

7.157 `write_csv_from_table` - Write CSV From Table

`write_csv_from_table(filename, table)`

Write `table` to disk as a Comma-Separated Value file called `filename`.
Equivalent to `write_delimited_file_from_table(",", filename, table)`.

7.158 `write_delimited_file_from_table` - Write Delimited File From Table

`write_delimited_file_from_table(delimiter, filename, table)`

Write `table` to disk as `filename` using newlines to separate rows and `delimiter` to separate columns. The file will be written with a header row corresponding to the keys of the columns of `table`. Returns the number of bytes written to the file.

```
KeRF> t: {{a: 1 2 3, b:["one", "two", "three"]}};
KeRF> write_delimited_file_from_table("|", "example.psv", t)
  23
KeRF> system("wc -c example.psv")
      23 example.psv
KeRF> system("cat example.psv")
a|b
1|one
2|two
3|three
```

7.159 `write_text` - Write Text

`write_text(filename, x)`

Write the value `x` to a text file `filename`, creating the file as necessary. If `x` is not already a string it will be converted to one as by `json_from_kerf`. Returns the number of bytes written to the file.

```
KeRF> write_text("example.txt", 5)
  1
KeRF> write_text("example.txt", 99)
  2
KeRF> system("cat example.txt")
99
```

7.160 `write_to_path` - Write to Path

`write_to_path(filename, x)`

Write the value `x` to a binary file `filename`, creating the file as necessary. Binary files are serialized in a custom format understood by `read_from_path`. Returns 0 if the operation was successful.

```
KeRF> write_to_path("example.bin", 23 24 25)
  0
KeRF> system("wc -c example.bin")
      64 example.bin
KeRF> system("hexdump example.bin")
0000000 06 90 00 fe 01 00 00 00 03 00 00 00 00 00 00 00
0000010 17 00 00 00 00 00 00 00 18 00 00 00 00 00 00 00
```

```
0000020 19 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000040
KeRF> read_from_path("example.bin")
  [23, 24, 25]
```

7.161 xbar - XBar

`xbar(x, y)`

Equivalent to `x * floor y/x`.

7.162 xkeys - Object Keys

`xkeys(x)`

Produce a list of keys for a map, table or list `x`.

```
KeRF> xkeys 33 14 9
  [0, 1, 2]
KeRF> xkeys {a: 42, b: 49}
  ["a", "b"]
KeRF> xkeys {{a: 42, b: 49}}
  ["a", "b"]
```

7.163 xvals - Object Values

`xvals(x)`

Produce a list of values for a map or table `x`. If `x` is a list, produce a list of valid indices to `x`.

```
KeRF> xvals 33 14 9
  [0, 1, 2]
KeRF> xvals {a: 42, b: 49}
  [42, 49]
KeRF> xvals {{a: 42, b: 49}}
  [[42], [49]]
```

8 Combinator Reference

8.1 converge - Converge

Given a unary function, apply it to a value repeatedly until it does not change or the next iteration will repeat the initial value. Some functional languages refer to this operation as `fixedpoint`.

```
KeRF> {[x] floor x/2} converge 32
  0
KeRF> {[x] mod(x+1, 5)} converge 0
  4
KeRF> {[x] display x; mod(x+1, 5)} converge 3
3
4
0
1
2
  2
```

If a numeric left argument is provided, instead repeatedly apply the function some number of times:

```
KeRF> 3 {[x] floor x/2} converge 32
  4
KeRF> 3 {[x] x*2} converge 32
  256
KeRF> 5 {[x] join("A", x)} converge "B"
  "AAAAAB"
```

Applied to a binary function, `converge` is equivalent to `fold`.

8.2 deconverge - Deconverge

`deconverge` is similar to `converge`, except it gathers a list of intermediate results.

```
KeRF> {[x] floor x/2} deconverge 32
  [32, 16, 8, 4, 2, 1, 0]
KeRF> {[x] mod(x+1, 5)} deconverge 1
  [1, 2, 3, 4, 0]
KeRF> 3 {[x] floor x/2} deconverge 32
  [32, 16, 8, 4]
KeRF> 3 {[x] x*2} deconverge 32
  [32, 64, 128, 256]
```

Applied to a binary function, `deconverge` is equivalent to `unfold`.

8.3 `fold` - Fold

Given a binary function, apply it to pairs of the elements of a list from left to right, carrying the result forward on each step. Some functional languages refer to this operation as `foldl`.

```
KeRF> add fold 1 2 3 4
  10
KeRF> {[a,b] join(enlist a, b)} fold 1 2 3 4
  [[[1, 2], 3], 4]
```

Note that the function will not be applied if `folded` over an empty or 1-length list:

```
KeRF> {[a,b] out "nope"; a+b} fold [5]
  5
KeRF> {[a,b] out "nope"; a+b} fold 1 2
nope  3
```

It is also possible to supply an initial value for the fold as a left argument:

```
KeRF> 0 {[a,b] join(enlist a, b)} fold 1 2 3
  [[[0, 1], 2], 3]
KeRF> 7 {[a,b] out "yep"; a+b} fold 5
yep  12
```

The symbol \/ is equivalent to fold:

```
KeRF> add \/ 1 2 3
  6
KeRF> 7 add \/ 5
  12
```

Applied to a unary function, `fold` is equivalent to `converge`.

8.4 `mapback` - Map Back

Given a binary function, `mapback` pairs up each value of a list with its predecessor and applies the function to these values. The first item of the resulting list will be the first item of the original list:

```
KeRF> join mapback 1 2 3
  [1,
   [2, 1],
   [3, 2]]
```

A common application of `mapback` is to calculate deltas between successive elements of a list:

```
KeRF> - mapback 5 3 2 9
  [5, -2, -1, 7]
```

If a left argument is provided, it will be used as the previous value of the right argument's first value:

```
KeRF> 1 join mapback 2 3 4
  [[2, 1],
   [3, 2],
   [4, 3]]
```

The symbol \~ is equivalent to mapback:

```
KeRF> 0 != \~ 1 1 0 1 0 0 0
  [1, 0, 1, 1, 1, 0, 0]
```

8.5 mapdown - Map Down

Apply a unary function to every element of a list, yielding a new list of the same size. Some functional programming languages refer to this as simply map. mapdown can be used to achieve a similar effect to how atomic built-in functions naturally “push down” onto the values of lists.

```
KeRF> negate mapdown 2 -5
  [-2, 5]
KeRF> {[n] 3*n} mapdown 2 5 9
  [6, 15, 27]
```

Given a binary function and a left argument, mapdown pairs up sequential values from two equal-length lists and applies the function to these pairs. Some functional programming languages refer to this as zip, meshing together a pair of lists like the teeth of a zipper:

```
KeRF> 1 2 3 join mapdown 4 5 6
  [[1, 4],
   [2, 5],
   [3, 6]]
```

mapdown also works with maps and tables:

```
KeRF> count mapdown {a: 1 2, b: 3 4 5, c: 6}
  [2, 3, 1]
KeRF> reverse mapdown {{a: 1 2, b: 3 4}}
  [[2, 1], [4, 3]]
```

The symbol \= is equivalent to mapdown:

```
KeRF> {[n] 3*n} \= 2 5 9
  [6, 15, 27]
```

8.6 mapleft - Map Left

Given a binary function, apply it to each of the values of the left argument and the right argument, gathering the results in a list.

```
KeRF> 1 2 3 join mapleft 4
  [[1, 4],
   [2, 4],
   [3, 4]]
```

The symbol \> is equivalent to mapleft.

8.7 mapright - Map Right

Given a binary function, apply it to a left argument and each of the values of the right argument, gathering the results in a list. `mapright`, like `mapdown`, provides a way of "pushing a function down onto" data or overriding existing atomicity:

```
KeRF> 1 join mapright 2 3 4
  [[1, 2],
   [1, 3],
   [1, 4]]
```

`mapright` and `mapleft` can be used to take the `cartesian product` of two lists:

```
KeRF> 0 1 2 add 0 1 2
  [0, 2, 4]
KeRF> 0 1 2 add mapright 0 1 2
  [[0, 1, 2],
   [1, 2, 3],
   [2, 3, 4]]
```

The symbol `\<` is equivalent to `mapright`.

8.8 reconverge - Reconverge

Equivalent to `unfold`.

8.9 reduce - Reduce

Equivalent to `fold`.

8.10 refold - Refold

Equivalent to `unfold`.

8.11 rereduce - Re-Reduce

Equivalent to `unfold`.

8.12 unfold - Unfold

`unfold` is similar to `fold`, except it gathers a list of intermediate results. This can often provide a useful way to debug the behavior of `fold`.

```
KeRF> add unfold 1 2 3 4
  [1, 3, 6, 10]
KeRF> 100 add unfold 1 2 3 4
  [101, 103, 106, 110]
KeRF> {[a,b] join(enlist a, b)} unfold 1 2 3 4
  [1,
   [1, 2],
   [[1, 2], 3],
   [[[1, 2], 3], 4]]
```

The symbol `\\` is equivalent to `unfold`:

```
KeRF> add \\ 1 2 3
  [1, 3, 6]
KeRF> 7 add \\ 5
  [12]
```

Applied to a unary function, `unfold` is equivalent to `deconverge`.

9 Global Reference

9.1 Math

9.1.1 .Math.BILLION - Billion

Constant representing $\lfloor 10^9 \rfloor$.

9.1.2 .Math.E - E

Constant representing Euler's number. 2.71828182845904523536O2.

9.1.3 .Math.TAU - Tau

Constant representing 2π. 6.283185307179586476925 2.

9.2 Net

9.2.1 .Net.client - Client

During IPC execution, contains a constant representing the current client's unique handle.

```
KeRF> open_socket("localhost", "10101")
  6
KeRF> send_sync(6, ".Net.client")
  6
KeRF> .Net.client
  0
```

9.2.2 .Net.on_close - On Close

If defined, an IPC server will call this single-argument function with a client handle when that client closes its connection.

```
KeRF> .Net.on_close: {[x] out 'client closed: ' join (string x) join '\n'};
KeRF>
server: new connection from 127.0.0.1 on socket 6
client closed: 6

KeRF>
```

9.3 Parse

9.3.1 .Parse.strptime_format - Time Format

Specifies the format used for formatting and parsing dates from delimited files when the field specifier is Z. Builds directly on the standard C function strptime(), as defined in time.h.

By default, %d-%b-%y %H:%M:%S.