

THE KERF PROGRAMMING LANGUAGE

John Earnest

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:

`kerf.concerns@gmail.com`

August 28, 2024

Contents

1	Introduction	8
1.1	Background	8
1.2	Conventions	8
1.3	Using the REPL	9
1.3.1	Command-Line Arguments and Scripting	9
1.3.2	The REPL	10
1.4	Examples	13
2	Installation	14
2.1	Installing (Binary)	14
2.2	Installing (License Files)	14
2.3	Installing (Building from Source)	15
2.4	Adding Kerf to your Path	15
2.5	Startup Scripts	16
3	Terminology	17
3.1	Atomicity	17
3.2	Combinators	18
3.3	Matrix	19
3.4	Truthy	19
3.5	Valence	19
3.6	Vector	19
4	Datatypes	20
4.1	Numbers	20
4.2	Lists and Vectors	21
4.3	Strings and Characters	22
4.4	Timestamps	23
4.5	Maps, Tables and Atlases	25
4.6	Special Identifiers	28
4.7	Index, Enum and Zip	29
4.8	Type Coercion	30
5	Syntax	32
5.1	Expressions	32
5.2	Indexing	35
5.3	Assignment	36
5.4	Control Structures	38
5.4.1	Conditionals	38
5.4.2	Loops	39
5.4.3	Function Declarations	40
6	SQL	41
6.1	Scoping	41
6.2	INSERT	42
6.3	DELETE	45
6.4	SELECT	46
6.4.1	WHERE	49
6.4.2	GROUP BY	50
6.5	UPDATE	51

6.5.1	WHERE	52
6.5.2	GROUP BY	52
6.6	Joins	53
6.6.1	Left Join	54
6.6.2	Asof Join	55
6.7	Limiting	56
6.8	Ordering	56
6.9	Performance	57
7	Input/Output	58
7.1	General I/O	58
7.2	File I/O	59
7.3	Striped Files	60
7.4	Parceled tables	61
7.5	Network I/O (IPC)	62
7.5.1	Starting an HTTP Server	64
7.5.2	Backgrounding a Kerf Server	64
8	Foreign Function Interface	65
8.1	C from Kerf	65
8.2	Kerf from C	67
8.2.1	Kerf .a archive and .o object files	67
8.3	Reference Counting	68
8.3.1	Releasing	68
8.3.2	Retaining	69
8.4	Internal Representations	71
8.4.1	Integers	71
8.4.2	Floats	71
8.4.3	Characters	71
8.4.4	Stamps	71
8.4.5	Vectors	71
8.5	Attribute Flags	72
8.6	Native API	74
8.6.1	kerf_api_append - Append to List	74
8.6.2	kerf_api_call_dyad - Call Dyad	74
8.6.3	kerf_api_call_monad - Call Monad	75
8.6.4	kerf_api_call_nilad - Call Nilad	75
8.6.5	kerf_api_copy_on_write - Copy On Write	75
8.6.6	kerf_api_get - Kerf Get	76
8.6.7	kerf_api_init - Initialize the Kerf Process	76
8.6.8	kerf_api_interpret - Interpret String	77
8.6.9	kerf_api_len - Kerf Length	77
8.6.10	kerf_api_new_charvec - New Kerf String	78
8.6.11	kerf_api_new_float - New Kerf Float	78
8.6.12	kerf_api_new_int - New Kerf Integer	78
8.6.13	kerf_api_new_kerf - New Kerf Object	78
8.6.14	kerf_api_new_list - New Kerf List	79
8.6.15	kerf_api_new_map - New Kerf Map	79
8.6.16	kerf_api_new_stamp - New Kerf Timestamp	80
8.6.17	kerf_api_nil - Kerf Nil	80
8.6.18	kerf_api_release - Release Kerf Reference	80
8.6.19	kerf_api_retain - Retain Kerf Reference	80

8.6.20	kerf_api_set - Kerf Set	81
8.6.21	kerf_api_show - Show Kerf Object	81
8.7	The Kerf IPC Protocol (KIP)	82
8.7.1	Execution Type	82
8.7.2	Response Type	82
8.7.3	Display Type	83
8.7.4	Wire Size	83
8.7.5	Shard Size	83
8.7.6	Payload Size	83
9	Built-In Function Reference	84
9.1	abs - Absolute Value	84
9.2	acos - Arc Cosine	84
9.3	add - Add	84
9.4	and - Logical AND	84
9.5	append_table_from_csv - Append Table From CSV File	85
9.6	append_table_from_fixed_file - Append Table From Fixed-Width File	85
9.7	append_table_from_psv - Append Table From PSV File	85
9.8	append_table_from_tsv - Append Table From TSV File	85
9.9	ascend - Ascending Indices	85
9.10	asin - Arc Sine	86
9.11	asof_join - Asof Join	86
9.12	atan - Arc Tangent	86
9.13	atlas - Atlas Of	86
9.14	atom - Is Atom?	87
9.15	avg - Average	87
9.16	bars - Time Bars, Sample Buckets, etc.	88
9.17	between - Between?	89
9.18	btree - BTree	89
9.19	bucketed - Bucket Values	89
9.20	car - Contents of Address Register	89
9.21	cdr - Contents of Decrement Register	89
9.22	ceil - Ceiling	90
9.23	char - Cast to Char	90
9.24	checksum - Object Hashcode	90
9.25	close_socket - Close Socket	90
9.26	combinations - Combinations	91
9.27	compressed - Compressed Vector Of	91
9.28	cos - Cosine	92
9.29	cosh - Hyperbolic Cosine	92
9.30	count - Count	92
9.31	count_nonnull - Count Non-Nulls	92
9.32	count_null - Count Nulls	92
9.33	create_table_from_csv - Create Table From CSV File	93
9.34	create_table_from_fixed_file - Create Table From Fixed-Width File	93
9.35	create_table_from_psv - Create Table From PSV File	93
9.36	create_table_from_tsv - Create Table From TSV File	93
9.37	cross - Cartesian Product	94
9.38	deal - Deal	94
9.39	delete_keys - Delete Keys	95
9.40	descend - Descending Indices	95
9.41	dir_ls - Directory Listing	96

9.42	display - Display	96
9.43	distinct - Distinct Values	96
9.44	divide - Divide	96
9.45	dlload - Dynamic Library Load	97
9.46	dotp - Dot Product	97
9.47	drop - Drop Elements	97
9.48	emu_debug_mode - Toggle Bytecode Debugger	98
9.49	enlist - Enlist Element	98
9.50	enum - Enumeration	98
9.51	enumerate - Enumerate Items	98
9.52	equal - Equal?	99
9.53	equals - Equals?	100
9.54	erf - Error Function	100
9.55	erfc - Complementary Error Function	100
9.56	eval - Evaluate	100
9.57	except - Except	100
9.58	exit - Exit	101
9.59	exp - Natural Exponential Function	101
9.60	explode - Explode	101
9.61	extract - Extract From Table	102
9.62	filter - Filter	102
9.63	first - First	103
9.64	flatten - Flatten	103
9.65	float - Cast to Float	103
9.66	floor - Floor	104
9.67	format - Format String	104
9.68	format_stamp - Format Timestamp	105
9.69	greater - Greater Than?	107
9.70	greatereq - Greater or Equal?	107
9.71	has_column - Table Has Column?	107
9.72	has_key - Has Key?	108
9.73	hash - Hash	108
9.74	hashed - Hashed	108
9.75	help - Help Tool	109
9.76	ident - Identity	109
9.77	ifnull - If Null?	109
9.78	implode - Implode	109
9.79	in - In?	109
9.80	index - Index	110
9.81	indexed - Indexed	110
9.82	int - Cast to Int	110
9.83	intersect - Set Intersection	110
9.84	isnull - Is Null?	111
9.85	join - Join	111
9.86	json_from_kerf - Convert Kerf to JSON	111
9.87	kerf_from_json - Convert JSON to Kerf	112
9.88	kerf_type - Type Code	112
9.89	kerf_type_name - Type Name	113
9.90	last - Last	113
9.91	left_join - Left Join	113
9.92	len - Length	113

9.93	less - Less Than?	114
9.94	lesseq - Less or Equal?	114
9.95	lg - Base 2 Logarithm	114
9.96	lines - Lines From File	114
9.97	ln - Natural Logarithm	115
9.98	load - Load Source	115
9.99	log - Logarithm	115
9.100	lsq - Least Squares Solution	116
9.101	map - Make Map	116
9.102	match - Match?	116
9.103	mavg - Moving Average	117
9.104	max - Maximum	117
9.105	maxes - Maximums	117
9.106	mcount - Moving Count	117
9.107	median - Median	118
9.108	meta_table - Meta Table	118
9.109	min - Minimum	118
9.110	mins - Minimums	118
9.111	minus - Minus	119
9.112	minv - Matrix Inverse	119
9.113	mkdir - Create directory	119
9.114	mmax - Moving Maximum	119
9.115	mmin - Moving Minimum	119
9.116	mmul - Matrix Multiply	120
9.117	mod - Modulus	120
9.118	msum - Moving Sum	120
9.119	negate - Negate	121
9.120	negative - Negative	121
9.121	ngram - N-Gram	121
9.122	not - Logical Not	121
9.123	noteq - Not Equal?	122
9.124	now - Current DateTime	122
9.125	now_date - Current Date	122
9.126	now_time - Current Time	122
9.127	open_socket - Open Socket	123
9.128	open_table - Open Table	123
9.129	or - Logical OR	123
9.130	order - Order	123
9.131	out - Output	123
9.132	parse_float - Parse Float From String	124
9.133	parse_int - Parse Integer From String	124
9.134	parse_stamp - Parse Timestamp From String	124
9.135	part - Partition	125
9.136	permutations - Permutations	125
9.137	plus - Plus	125
9.138	pow - Exponentiation	126
9.139	powerset - Power Set	126
9.140	rand - Random Numbers	126
9.141	range - Range	127
9.142	read_from_path - Read From Path	127
9.143	read_parceled_from_path - Read Parceled Table From Path	128

9.144	read_stripped_from_path - Read Striped File From Path	128
9.145	read_table_from_csv - Read Table From CSV File	128
9.146	read_table_from_delimited_file - Read Table From Delimited File	128
9.147	read_table_from_fixed_file - Read Table From Fixed-Width File	130
9.148	read_table_from_tsv - Read Table From TSV File	130
9.149	rep - Output Representation	131
9.150	repeat - Repeat	131
9.151	reserved - Reserved Names	131
9.152	reset - Reset	131
9.153	reverse - Reverse	132
9.154	rsum - Running Sum	132
9.155	run - Run	132
9.156	search - Search	133
9.157	seed_prng - Set random seed	133
9.158	send_async - Send Asynchronous	133
9.159	send_sync - Send Synchronous	133
9.160	setminus - Set Disjunction	133
9.161	shell - Shell Command	133
9.162	shift - Shift	134
9.163	shuffle - Shuffle	134
9.164	sin - Sine	134
9.165	sinh - Hyperbolic Sine	135
9.166	sleep - Sleep	135
9.167	sort - Sort	135
9.168	sort_debug - Sort Debug	136
9.169	split - Split List	136
9.170	sqrt - Square Root	136
9.171	stamp - Cast to Stamp	137
9.172	stamp_diff - Timestamp Difference	137
9.173	std - Standard Deviation	137
9.174	string - Cast to String	137
9.175	subtract - Subtract	137
9.176	sum - Sum	138
9.177	table - Make Table	138
9.178	tables - Tables	138
9.179	take - Take	139
9.180	tan - Tangent	139
9.181	tanh - Hyperbolic Tangent	139
9.182	times - Multiplication	140
9.183	timing - Timing	140
9.184	tolower - To Lowercase	140
9.185	toupper - To Uppercase	140
9.186	transpose - Transpose	141
9.187	trim - Trim	141
9.188	type_null - Type Null	141
9.189	uneval - Uneval	142
9.190	union - Set Union	142
9.191	unique - Unique Elements	142
9.192	unzip - Decompress Object	142
9.193	var - Variance	142
9.194	which - Which	143

9.195	write_csv_from_table - Write CSV From Table	144
9.196	write_delimited_file_from_table - Write Delimited File From Table	144
9.197	write_striped_to_path - Write Striped File To Path	144
9.198	write_text - Write Text	144
9.199	write_to_path - Write to Path	144
9.200	xkeys - Object Keys	145
9.201	xvals - Object Values	145
9.202	zip - Compress Object	145
10	Combinator Reference	146
10.1	converge - Converge	146
10.2	deconverge - Deconverge	146
10.3	fold - Fold	147
10.4	mapback - Map Back	147
10.5	mapcores - Map to Cores	148
10.6	mapdown - Map Down	149
10.7	mapleft - Map Left	149
10.8	mapright - Map Right	149
10.9	reconverge - Reconverge	150
10.10	reduce - Reduce	150
10.11	refold - Refold	150
10.12	rereduce - Re-Reduce	150
10.13	unfold - Unfold	150
11	Global Reference	151
11.1	Environment	151
11.1.1	.Argv - Arguments	151
11.1.2	.Help - Function Reference	151
11.2	Math	151
11.2.1	.Math.BILLION - Billion	151
11.2.2	.Math.E - E	151
11.2.3	.Math.TAU - Tau	151
11.3	Net	151
11.3.1	.Net.client - Client	151
11.3.2	.Net.on_close - On Close	151
11.3.3	.Net.parse_request - Parse Request	151
11.4	Parse	152
11.4.1	.Parse.strptime_format - Time Stamp Format	152
11.4.2	.Parse.strptime_format2 - Time Format	152
11.5	Print	152
11.5.1	.Print.stamp_format - Print Stamp Format	152
12	Programming Techniques	153
12.1	Reversing a Map	153
12.2	Run-Length Encoding	156
12.3	Base64 Conversion	160
12.4	HTTP Fetching	166
12.5	Kerf IPC with Python	172
12.5.1	HudsucKerf By Proxy	175

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 SQL syntax. 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 Background

The Kerf team was first introduced to array languages in 2006 by Dennis Shasha at NYU. This led to work with Arthur Whitney's & Kx System's kdb+ family of languages at the investment banks Cantor Fitzgerald and Merrill Lynch. A precursor language to Kerf called Kona was started around 2009. Kona was open-sourced in Summer 2010 and improved upon with the community for the following four years. Lessons from Kona would inspire Kerf.

The first lines of code for Kerf were written in Summer 2014. Kerf officially launched as a product in Spring 2015. Kerf is the team's third major programming language release, and arguably a fifth generation language and database system. It draws on lessons from over twenty years of programming. It is mature technology.

The name "Kerf" comes from a term in woodworking- the cut made by a saw. It springs from the Old English "cyrf", the action of cutting. It's short, strong, and simple.

1.2 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
```

Sometimes examples will contain comments, colored orange to help set them apart from code:

```
2+3;    // kerf uses c-style line comments
```

1.3 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 you have installed Kerf in your path, you can simply type `kerf`. The rest of this discussion will assume the latter case. To exit the REPL use the key combination `Ctrl+d`.

1.3.1 Command-Line Arguments and Scripting

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
<code>-q</code>	-	Suppress the startup banner.
<code>-l</code>	-	Enable debug logging.
<code>-e</code>	String Expression	Execute an expression.
<code>-x</code>	String Expression	Evaluate an expression and print the result.
<code>-p</code>	Port Number	Specify a listening port for starting an IPC server.
<code>-P</code>	Port Number	Specify a listening port for starting an HTTP server.
<code>-V</code>	Number of Bytes	Cap virtual memory used for disk-backed storage.
<code>-R</code>	Number of Bytes	Cap physical workspace RAM reserved by Kerf.

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 filenames as command-line arguments, the contents of those files 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'"
```

The flags `-V` and `-R` permit configuring how Kerf uses system memory. By default, there is no cap for workspace ram usage, and virtual memory for disk-backed tables is also *effectively* uncapped. We can easily observe the behavior of limiting usable workspace RAM:

```
> ./kerf -q -R 10000
KeRF> range(1, 500)
[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, ...]
KeRF> range(1, 20000)
User-set memory bound prevented further memory allocation.

  range(1, 20000)
  ^
Virtual memory error
KeRF>
```

Note that this limit applies only to Kerf's reference-counted internal pools; external libraries and IPC code can allocate additional memory. Enforcing harder limits will require operating-system-specific tools.

1.3.2 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]]
```

The up and down cursor keys can be used to cycle through recently entered commands, saving repetitive typing. For your convenience, this history information is stored in `$HOME/.kerf_history` across sessions.

The key combination Control-d will cause Kerf to exit by sending an EOF to the shell. For many this is the preferred way to exit the REPL.

If your program enters an infinite loop or otherwise seems to have locked up, Pressing Control-c will interrupt execution, allowing you to make corrections. Control-z is a stronger epithet which will stop execution of the Kerf process and move it to the background of the shell. Note that backgrounded processes are still alive and still consume memory. For more information regarding stopped processes, refer to Unix documentation for the `jobs`, `fg` and `kill` commands.

```
KerF> for(i: 0; i < 10; i: i-1) {} // whoops.  
^C  
  for(i: 0; i < 10; i: i-1)  
    ^  
Caught interrupt signal
```

During a session you can use **reset** to clear the workspace and close all open resources. When you're done using Kerf, the **exit** function will exit the REPL and return you to your shell. You may also use the key combination Control-D to exit Kerf.

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

Kerf also special-cases a number of common commands to save you typing:

- **exit** - equivalent to calling **exit(0)**. Ends the session and stops the Kerf process.
- **reset** - equivalent to calling **reset()**. Resets the interpreter and workspace, preserving command line flags to the original Kerf process.
- **clear** - equivalent to calling **system('clear')**. Clear the terminal output.
- **help** - equivalent to calling **help()**. Display the top-level index for the built-in Kerf command reference.

```
KeRF> help
```

Help Menu. Try: `help('list')`.

Press return to see the next part of the table.

subject
list
strings
table
aggregate
math
combinator
sql
misc
...

```
KeRF>
```

1.4 Examples

Let's look at a few short Kerf snippets to get a taste of the language:

Are two strings anagrams?

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

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

Gather simple statistics for random data using SQL syntax:

```
KeRF> data: {{a: rand(100, 5)}};  
KeRF> SELECT count(a) AS items, avg(a) AS average FROM data
```

items	average
100	1.82

Load a text file and interactively query it:

```
KeRF> characters: tolower flatten lines "flour.txt"  
"flour is a powder made by grinding uncooked cereal grains or other seeds or roots (like  
cassava). it is the main ingredient of bread, which is a staple food for many cultures,  
making the availability o..."  
KeRF> sum characters in "aeiou"  
182  
KeRF> count characters  
540  
KeRF> 5 take ` " " explode characters  
["flour", "is", "a", "powder", "made"]
```

Iteratively calculate terms of the Fibonacci sequence without using explicit loops:

```
KeRF> 6 {[x] last(x) join sum x} deconverge 1 1  
[[1, 1], [1, 2], [2, 3], [3, 5], [5, 8], [8, 13], [13, 21]]  
KeRF> first mapdown 10 {[x] last(x) join sum x} deconverge 1 1  
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

2 Installation

Pre-compiled evaluation copies of Kerf for 64-bit Linux and OSX can be obtained by request:

`kerf.concerns@gmail.com`

2.1 Installing (Binary)

Kerf binaries 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.

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

2.2 Installing (License Files)

Commercial installations of Kerf typically use a licensing file `kerf-license.dat`. It is recommended that this license file be placed in `~/.kerf/kerf-license.dat`, where `~/.kerf` is the hidden `.kerf` directory under the users's home directory.

If Kerf does not find the license file in `~/.kerf` it will look in the current working directory (`cwd`) as a fallback. This is usually less convenient.

When the time has expired on the license file it will need to be replaced. If you have an active link to the license server where a license file can be downloaded, you can download a new license file to replace the old one, and the expiration period will refresh.

```
/home/john > cd $HOME                #change to your home directory
/home/john > mkdir -p $HOME/.kerf    #create the directory if it does not exist
/home/john > cp kerf-license.dat $HOME/.kerf/kerf-license.dat #copy the license file
```

2.3 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. Compiling from source will also produce a binary named `kerf_test` which executes self-tests at startup and executes in debug mode, printing extra information when errors are encountered. The debug mode binary is particularly helpful when debugging dynamic libraries, as it can detect many types of memory leak at shutdown.

2.4 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 a line to initialize this setting.

If you're using `bash` (The default shell on OSX):

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

If you're using `tcsh` or `csh` (The default shell in some Linux distros):

```
set path = ($path /opt/kerf)
```

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

2.5 Startup Scripts

While working with Kerf, you may find yourself writing reusable utility routines. For example, here's a simple predicate which returns true if a timestamp falls on the current day, and then a function which filters lists based on this predicate:

```
def today(s) {
    t: ["year", "month", "day"]
    return match(now()[t], s[t])
}

def on_today(v) {
    return today filter v
}
```

```
KeRF> on_today 2016.01.11 2016.01.12 2016.01.13
[2016.01.12]
```

If you use these routines often, you might want them to be automatically loaded when you open the Kerf REPL. Kerf will search for a file named `startup.kerf`, first in a directory given by the environment variable `KERF_HOME`, then in a `.kerf` directory in the user's `HOME` directory, and finally in the current directory. Your startup script can in turn execute additional files by using `load`.

```
> cat ~/kerf_home/startup.kerf

def calendar() {
    out "\n" implode shell "cal"
}

out "Loaded custom startup script.\n"

> export KERF_HOME=$HOME/kerf_home/
> echo $KERF_HOME
/Users/john/kerf_home/

> kerf -q
Loaded custom startup script.

KeRF> calendar()
    January 2016
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
31
KeRF>
```

To set `KERF_HOME` persistently, add an appropriate line to `.profile`, just as you did to add the `kerf` executable to your `PATH`.

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]

KerF> add(1 2 3, 4 5)
add(1 2 3, 4 5)
^
Length error
```

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 `folding` a user-declared lambda, it must construct and then reduce the entire list.

3.3 Matrix

A *matrix* is a *vector* of *vectors* 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 Truthy

Kerf does not have a special “boolean” type for representing the values *true* and *false*. By convention, the values 1 and 0 are used in most cases- this is particularly helpful in combination with the **which** operator:

```
Kerf> [true, false, true, true]  
      [1, 0, 1, 1]  
Kerf> which 1 0 1 1  
      [0, 2, 3]
```

In some situations, Kerf permits a broader range of values to behave like true, or *truthy*. Values which behave like false are naturally *falsey*. Any numeric zero or null is falsey, and any other value is considered truthy.

```
Kerf> if (-5) { display "yep" }  
"yep"  
Kerf> if ([]) { display "yep" }  
"yep"  
Kerf> if ({} ) { display "yep" }  
"yep"  
Kerf> if (NaN) { display "yep" }  
"yep"  
Kerf> if (0) { display "nope" }  
Kerf> if (0.0) { display "nope" }  
Kerf> if (null) { display "nope" }
```

Heavy reliance on truthiness can lead to very confusing code. Prefer 1 and 0, or the literals true and false, whenever possible.

3.5 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.

3.6 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 $2^{63} - 1$.

$$\begin{array}{r} 42 \\ 010 \\ +976 \\ -9000 \end{array}$$

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:

[illegible]

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
```

Numeric values may be cast to integers or floats by using the built-in functions `int` and `float`, respectively. Strings can be parsed as numbers by the functions `parse_int` and `parse_float`, or in other cases by simply using `eval`.

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 function **char** can convert a number into an equivalent character, and **int** will convert characters into numeric character codes:

```
KerF> char 65 66 67
"ABC"
KerF> int "Text"
[84, 101, 120, 116]
```

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

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

Note that a single character in quotation marks is a *string*, not a character literal:

```
KerF> kerf_type_name "A"
"character vector"
KerF> "A" match `"A"
0
KerF> first "A"
`"A"
```

When dealing with characters, spaces are considered null values:

```
KerF> isnull ` " "
1
KerF> (not isnull) filter "Text with whitespace! "
"Textwithwhitespace!"
```

To assemble strings from other assorted datatypes, see **format**.

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**.

```
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 function **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)
1728000000000000
```

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"]
2016.01.06
KeRF> d[["date", "time"]]
[2016.01.06, 16:54:52.021]
KeRF> d[["year", "month", "week", "day", "hour", "minute", "second", "millisecond", "nanosecond"]]
[2016, 1, 2, 6, 16, 54, 52, 21, 21124000]
```

Strings can be converted to stamps by using **eval**, and **rep** or **string** can reverse the process:

```
KeRF> eval "2001.10.10"
2001.10.10
KeRF> rep 2001.10.10
"2001.10.10"
KeRF> string 2001.10.10
"2001.10.10T00:00:00.000"
```


It is also possible to use the **stamp** function:

```
KerF> stamp ["2001.02.03", "1985.08.17"]  
[2001.02.03, 1985.08.17]
```

For more flexibility, see **parse_stamp** and **format_stamp**. These functions support a superset of the parsing and formatting capabilities of the standard C `strftime()` and `strptime()` functions which can handle milliseconds or nanoseconds:

```
KerF> parse_stamp("%S:%M:%H", "56:34:12")  
12:34:56.000  
KerF> format_stamp("%H:%M:%S.%q", now())  
"00:43:54.152979000"
```

Timestamps are currently valid through 2262.04.11, at which point we'll already have transitioned to 128-bit operating systems. Be sure to download the latest Kerf release when nearing the “Year 2.262k Problem”.

4.5 Maps, Tables and Atlases

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. In any of these cases, the keys of the resulting map will be strings.

```
{ "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}}                 // columns contain [] and []
```

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. The builtin **map** produces a map from a list of keys and a list of values, and **table** works similarly for a list of column names and a rectangular array of values.

```
KerF> xkeys {a: 10 11 12, b: 20 21}
["a", "b"]
KerF> xvals {{a:1, b:2}}
[[1], [2]]
KerF> map(["a","b"], [37, 99])
{a:37, b:99}
KerF> table(["first","second"],[1 2, 3 4])
```

first	second
1	3
2	4

delete_keys can remove entries from maps or tables:

```
KerF> delete_keys({a:1, b:2, c:3}, ["c"])
{a:1, b:2}
KerF> delete_keys({{a:1, b:2, c:3}}, ["b","a","b","x"])
```

c
3

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

Note that maps are treated by some built-in functions as atoms. For example, **len** considers a map with any number of key-value associations to be of length 1. To obtain the number of associations in a map, use **len xvals** or **len xkeys**. If you're really desperate to save keystrokes you can use **|^ (len enumerate)**.

```
KeRF> len {}
1
KeRF> len {a:2, quux:"a string"}
1
KeRF> len xkeys {}
0
KeRF> len xkeys {a:2, quux:"a string"}
2
KeRF> |^{a:1, b:2}
2
```

If this behavior for maps is confusing, consider that with a table **len** indicates the number of rows, rather than the number of columns. A map is behaving just like a table with a single row:

```
KeRF> len {{a:0 1 2 3, b:4 5 6 7}}
4
KeRF> len {{a:0, b:4}}
1
```

An *Atlas* behaves somewhat like a table with varying keys, for unstructured or semi-structured data. Atlases are sets of maps enclosed in one set curly braces with a square brace (**{[** and **]**). Atlases can also be constructed from lists of maps using the **atlas** function. The Atlas type is to be used for unstructured or semi-structured data, for example, financial forms with varying columns.

```
KeRF> [{a: 1, b: 3},{a: 3, c: 2}] // simple atlas

KeRF> atlas([a:1,b:3],a:3,c:2]) // same atlas using atlas constructor

KeRF> [{a: 1, b: 2},{a:2,d:{z:1,e:7}}] // nested atlas
```

Atlases, unlike simple key-value stores, can be queried and aggregated in ways similar to tables, despite not having consistent columns. Aggregates or selected keys always return a table. Otherwise, an atlas is returned.

```
KeRF> atla: atlas([a:1,b:2},{b:4,c:5},{a:3,b:3,d:9}]);select avg(a) from atla
```

a
2.0

```
KeRF> select avg(b) from atla
```

b
3.0

```
KeRF> select from atla where b=4
atlas[{b:4, c:5}]
```

```
KeRF> select a,b from atla where b>2
```

a	b
NAN	4
3	3

Nested keys can also be accessed.

```
KeRF> atla: {[ {a:1,b:2}, {b:4,c:5}, {b:{d:6,e:7}} ]}
```

```
KeRF> select b.e from atla where b.d=6
```

b.e
7

Finally, group by is also supported

```
KeRF> atla: {[ {c:10,b:4,a:1}, {b:4,c:10,a:2}, {c:11, b:5,a:3} ]}
```

```
KeRF> select sum(a) from atla group by c where a>=2
```

c	a
10	2
11	3

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 interchangeably 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
Undefined token error
KerF> root
{a:24}
```

4.7 Index, Enum and Zip

Indexes, Enumerations and Zips 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
"btree sort"
```

A *Zip* is a list which is stored in memory in a compressed form, making it easier to work with large datasets. There are a number of specialized compressed list subtypes- see the discussion in **compressed**.

4.8 Type Coercion

Some operators can be applied to arguments of several distinct types. For example, **add** accepts both integers and floats, and will work if given one of each as arguments:

```
KeRF> 1 add 2.5
3.5
```

In situations like these, the Kerf interpreter follows simple rules to *coerce* arguments to different types and determine the appropriate return type.

In numeric operations, integer values will be converted to floats. For very large values this can lose some precision; 64-bit floats represent *fewer* distinct values than 64-bit integers, distributed over a larger range:

```
KeRF> 1.0 + 100000000000000002
100000000000000000.0
```

Similarly, a float vector combined with an integer vector will promote the integer vector to a float vector and yield a float vector as a result:

```
KeRF> kerf_type_name a:2.1 3.0 4.0
"float vector"
KeRF> kerf_type_name b:3 5 7
"integer vector"
KeRF> a+b
[5.1, 8, 11.0]
KeRF> kerf_type_name a+b
"float vector"
```

If you append an integer to a float vector, it remains a float vector. Appending a float to an integer vector promotes the result to a float vector:

```
KeRF> kerf_type_name join(0.5 0.6, 3)
"float vector"
KeRF> kerf_type_name join(0.2, 5 6)
"float vector"
```

An empty list combined with a vector or vectorizable type (integers, floats, timestamps or characters) will always produce a vector:

```
KeRF> kerf_type_name []
"list"
KeRF> kerf_type_name join([], 2)
"integer vector"
KeRF> kerf_type_name join([], 3.5 3.6)
"float vector"
```

Lists *may* automatically become vectors when modified to contain items of a uniform vectorizable type. The behavior of specific operators in this manner is an optimization, and may change with future revisions of Kerf. Try not to depend on this behavior, and avoid using mixed-type lists when you want the performance benefits of vectors:

```
KerF> kerf_type_name a: ["foo", 12, 3.5]
"list"
KerF> a: drop(1, a)
[12, 3.5]
KerF> kerf_type_name a
"float vector"
```

If you have a list which is known to only contain numeric values, you can *explicitly* convert it to a vector by using the built-in functions **int** or **float**:

```
KerF> kerf_type_name a: xvals {a:0.4, b:4}
"list"
KerF> int a
[0, 4]
KerF> kerf_type_name int a
"integer vector"
KerF> float a
[0.4, 4.0]
KerF> kerf_type_name float a
"float vector"
```

When combining lists or vectors with indexes or enumerations, the properties of the left argument will be preserved. Kerf will never spontaneously promote vectors or lists to indexes or enumerations.

```
KerF> join(["foo", "bar"], ["quux"])
#["foo", "bar", "quux"]
KerF> join(["quux"], #["foo", "bar"])
["quux", "foo", "bar"]
KerF> join(=[3,7,2,1], 4)
=[3, 7, 2, 1, 4]
KerF> join(4, =[3,7,2,1])
[4, 3, 7, 2, 1]
```


5 Syntax

5.1 Expressions

Calling (or *applying*) a function in Kerf resembles most conventional 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> times(3, 5)
15
KerF> 3 + 5
8
KerF> 3 plus 5
8
KerF> plus(3, 5)
8
```

That said, note that the symbolic operators do not bind with parentheses in the same way as the textual operators. Here the parenthetical expression is *not* evaluated as an argument, and so returns merely the last item in the expression, which is then operated on by the symbolic operator.

```
KeRF> -(3, 5)
-5
```

Operators have uniform 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
```

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

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 look 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]
22
```

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:

```
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]
```

Ordinarily, referencing an uninitialized variable is an error. In some situations, including compound assignment, the variable will instead act as though initialized with an empty map. Try to avoid depending on this behavior:

```
KeRF> x
      Undefined token error
KeRF> x#:2
      [{}, 2]
```

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. 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
```

In Kerf, newlines are statement separators. Conditional statements spread across multiple lines must adhere to a specific, consistent indentation style:

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

Multiline conditionals *must not* be written with a newline before `else if` or `else` clauses. Implied (and undesirable) statement separators are shown in **red**:

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

Another incorrect indentation style:

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

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; 43 }  
42  
42  
42  
43  
  
KerF>
```

If it is necessary to prematurely exit a loop, break the loop into its own function and use return.

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:

```
x: 35  
function outer_1() {  
    x: 25  
    function inner() {  
        return x  
    }  
    return inner;  
}  
function outer_2() {  
    x: 15  
    l: outer_1()  
    x: 45  
    return l()  
}  
display outer_2()
```

This example will print 25, because `inner` captures the definition of `x` in `outer_1` when it is created. The definitions of `x` in `outer_2` are not used when this function is evaluated, nor is the top-level definition of `x`.

6 SQL

Kerf understands 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, but for clarity 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 Scoping

When the target of a write like INSERT or UPDATE is a variable name, the variable is resolved as if it were in the global scope, even when the statement occurs inside of a function. This is done for convenience: database code is typically written like this, and this is the most common use case. To use INSERT or UPDATE as if they were functional methods, either parenthesize your variable or wrap it in the `ident` function, then save the result into another local variable.

```
insert into t values b:4,c:5          //t is affected in the global scope. no save necessary  
t: insert into (t) values b:4,c:5    //a copy of the local t is modified and saved, functionally
```

6.2 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
```

The value **table** can be an object such as table literal, or it can be the name of a variable containing a table. In the latter case, the table will be modified in-place. The value **data** can be a list, matrix, map or table.

To insert a row, use an ordinary square-bracketed Kerf list. You can also use a map, which more explicitly shows column names in the source data. For bulk inserts, you can insert a table.

```
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

```
KeRF> INSERT INTO {{{}} VALUES {{name:["bob","jim"], level:7 8, email:["b@ob.com","j@im.com"]}}
```

name	email	level
bob	b@ob.com	7
jim	j@im.com	8

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
```

The examples above show INSERT for creating tables. The more common use case involves inserting into an already existing table. For clarity sake:

```
KeRF> t:{{a:1 2,b:3 4,c:99.9 10}}
```

a	b	c
1	3	99.9
2	4	10.0

```
KeRF> INSERT INTO t VALUES {a:99, b:-10, c:1.1}
```

a	b	c
1	3	99.9
2	4	10.0
99	-10	1.1

```
KeRF> INSERT INTO t VALUES [[8,9],[44, 55],[82.1, -8.8]]
```

a	b	c
1	3	99.9
2	4	10.0
99	-10	1.1
8	44	82.1
9	55	-8.8

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

6.3 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

6.4 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

SELECT can also be used to reorder, duplicate, or add columns:

```
KeRF> SELECT gender, age, age AS years FROM people
```

gender	age	years
M	37	37
F	29	29
M	48	48
F	31	31
F	16	16
M	29	29
M	18	18
F	29	29

```
KeRF> other: {{ b: range len people }}
```

b
0
1
2
3
4
5
6
7

```
KeRF> SELECT other.b AS id, * FROM people
```

id	name	age	gender	job
0	Hamilton Butters	37	M	Janitor
1	Emma Peel	29	F	Secret Agent
2	Jacques Maloney	48	M	Private Investigator
3	Renee Smihee	31	F	Programmer
4	Karen Milgram	16	F	Student
5	Chuck Manwich	29	M	Janitor
6	Steak Manhattan	18	M	Secret Agent
7	Tricia McMillen	29	F	Mathematician

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

When working with the results of a **SELECT** query, sometimes you don't actually want a table- you just want a single result. You can destructure tables explicitly with the primitives **xkeys** and **xvals**. Alternatively, use the more convenient context-sensitive function **extract**, which turns single-column tables into an appropriate atom or list:

```
KeRF> SELECT min(age) FROM people
```

age
16

```
KeRF> extract SELECT min(age) FROM people
16
```

6.4.1 WHERE

The WHERE clause permits filtering of results. Only rows which adhere to the constraints given as the **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 conjunction 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

6.4.2 GROUP BY

The GROUP BY clause can be used to gather together sets of rows which match on a particular column. 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

6.5 UPDATE

UPDATE modifies a table in-place, altering the values of some or all columns of rows which match a query.

```
UPDATE table SET assignments
    [ WHERE condition ]
    [ GROUP BY aggregate ]
```

In its simplest form, UPDATE transforms or reassigns one or more of the columns of the table. The right side of each clause of **assignments** can be any Kerf expression. Note that the SET clause can use the symbols = or : to represent assignment, for compatibility with familiar SQL engines. In practice, favor using : to avoid ambiguity.

```
KeRF> UPDATE {{a:10 20 30}} SET a=a*2
```

a
20
40
60

```
KeRF> UPDATE {{a:10 20, b: 40 50}} SET a:11+a, b=5
```

a	b
21	5
31	5

```
KeRF> UPDATE {{a:0 1 2, b: 3 4 5}} SET b:5
```

a	b
0	5
1	5
2	5

Assignments are carried out left to right:

```
KeRF> UPDATE {{a:"First", b:"Second"}} SET a=b, b=a
```

a	b
Second	Second

6.5.1 WHERE

An UPDATE can be restricted to only modify a specific subset of rows by using a WHERE clause, as in SELECT:

```
KeRF> UPDATE {{a:0 1 2, b:7 3 7}} SET b=99 WHERE b=7
```

a	b
0	99
1	3
2	99

6.5.2 GROUP BY

UPDATE also supports the GROUP BY clause, operating on sets of rows which match on a particular column:

```
KeRF> t: {{a: 0 1 2 3 4 5, b: 0 1 1 0 2 3, c: 0 0 0 0 0 0}};
```

```
KeRF> UPDATE t SET c=count(b);
```

```
KeRF> t
```

a	b	c
0	0	6
1	1	6
2	1	6
3	0	6
4	2	6
5	2	6

```
KeRF> UPDATE t SET c=count(b) GROUP BY b;
```

```
KeRF> t
```

a	b	c
0	0	2
1	1	2
2	1	2
3	0	2
4	2	1
5	3	1

```
KeRF> UPDATE t SET c=enlist(a) GROUP BY b;
```

```
KeRF> t
```

a	b	c
0	0	[0, 3]
1	1	[1, 2]
2	1	[1, 2]
3	0	[0, 3]
4	2	[4]
5	3	[5]

UPDATE...GROUP BY can also modify columns in place:

```
KeRF> t: {{a: 99 99 37 99 37 479}};
KeRF> UPDATE t SET a=count(a) GROUP BY a;
KeRF> t
```

a
3
3
2
3
2
1

6.6 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

6.6.1 Left Join

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 (z). 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

6.6.2 Asof Join

Behaves as `left_join` for the first three arguments. The fourth argument 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

6.7 Limiting

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

```
KeRF> first(2, SELECT name, age FROM people WHERE name = "Emma Peel")
```

name	age
Emma Peel	29
Emma Peel	29

A better approach is to define a new function which takes the minimum of n and the length of the result:

```
KeRF> limit_rows: {[n, t] first(min(count t, n), t)};
```

```
KeRF> limit_rows(2, SELECT name, age FROM people WHERE name = "Emma Peel")
```

name	age
Emma Peel	29

6.8 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

6.9 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 < 23.7};

KerF> timing 1;
KerF> find v;
      15 ms
KerF> find iv;
      6 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 Input/Output

Kerf provides a rich set of built-in IO functions for displaying, serializing, importing and exporting data. Beyond the capabilities described here, Kerf provides a general purpose foreign-function interface- see **FFI**.

7.1 General I/O

The function **out**(x) will print a string x to standard output. Non-string values are ignored:

```
KerF> out "foo"
foo
KerF> out 65
KerF>
```

The function **display**(x) will print a display representation of data to standard output. 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]
```

The function **shell**(x) will execute a string x containing a shell command as if from `/bin/sh -c x` and return the lines of the result:

```
KerF> out implode("\n", shell("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> shell("echo Hello, World!")
["Hello, World!"]
```

7.2 File I/O

The function `dir_ls` lists the files and directories at a path. It can optionally provide full path 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"]
```

The function `lines(filename, n)` loads lines from a plain text file into a list of strings. The argument `n` is optional, and specifies the maximum number of lines to read.

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

The function `write_text(filename, x)` writes a raw string to a file, returning the number of bytes written. If `x` is not already a string it will be converted to one as by `json_from_kerf`. To perform the inverse of `lines`, use `write_text(filename, implode("\n", x))`.

```
KerF> write_text("example.txt", 5)
1
KerF> write_text("example.txt", 99)
2
KerF> shell("cat example.txt")
["99"]
```

Kerf has a proprietary binary serialization format. The function `write_to_path(filename)` writes a Kerf object to a file, creating it if necessary, and returns 0 if the operation was successful. Objects can then be reconstituted from binary files by calling `read_from_path(filename)`.

```
KerF> write_to_path("example.bin", 23 24 25)
0
KerF> shell("wc -c example.bin")
["    64 example.bin"]
KerF> out implode("\n", shell("hexdump example.bin"))
00000000 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]
```

Kerf also has built-in functions which make it easy to read and write common tabular data formats. `read_table_from_delimited_file` loads data, `write_delimited_file_from_table` writes data, and several wrapper functions are available which read and write CSV or TSV files using this functionality.

Kerf can load data from text files with fixed-width columns with `read_table_from_fixed_file`. See the built-in function reference discussion of this function for additional details.

7.3 Striped Files

Kerf is also capable of working with “striped” files. This serialization format writes each nested component of an object (the items of a list, or the columns of a table, for example) as a separate file within a directory structure. Striped files are more space-efficient for sparse structures, and permit significantly faster appends.

To create a striped file from an object, use `write_striped_to_path(path, x)`:

```
KerF> write_striped_to_path("stripe_example", {{a: 1 2 3, b: 4 5 6}});
KerF> out "\n" implode shell "tree stripe_example"
stripe_example
├── 2
│   ├── 0
│   │   └── base.dat
│   ├── 1
│   │   └── base.dat
│   └── base.dat
└── base.dat

3 directories, 4 files
```

To load a striped file, use `read_striped_from_path(path)`:

```
KerF> read_striped_from_path("stripe_example")
```

a	b
1	4
2	5
3	6

For high-performance data storage, Kerf provides a set of alternatives to the “read_table_”-prefixed methods which begin with “create_table_”, like `create_table_from_csv`. Instead of simply reading a file into memory, these routines load data into a memory-mapped, disk-backed data structure, deleting any pre-existing destination file if necessary. These routines will use striped files preferentially, but also support conventional serialized Kerf objects. The `create_table_` functions allow you to work efficiently with large data files which may not fit in memory at once.

There are also a series of “append_table_”-prefixed equivalents which, as you might expect, append data to any pre-existing destination file or create a new file if none exists.

7.4 Parceled tables

A natural way of storing large amounts of time series data is parceling the data by day into separate subdirectories which are named by day. This allows for easier updates, backups and error recovery. Kerf provides a table type which logically connects the separate daily parcels. The requirements are that the subdirectories are named by the correct days. In the following example, we will create a small table with two days worth of data, create a directory in the /tmp directory, write out the tables with the data from the given days, then read the table in as a parceled table using `read_parceled_from_path(path):`.

```
KerF> twodays: {{timedate: {[x] 1999.01.01 + 8h*x } mapright range(6), vals: range(6)}}
```

timedate	vals
1999.01.01	0
1999.01.01T08:00:00.000	1
1999.01.01T16:00:00.000	2
1999.01.02	3
1999.01.02T08:00:00.000	4
1999.01.02T16:00:00.000	5

```
KerF> mkdir("/tmp/p");
```

```
KerF> write_stripped_to_path("/tmp/p/1999.01.01",select * from twodays where timedate <1999.01.02);
```

```
KerF> write_stripped_to_path("/tmp/p/1999.01.02",select * from twodays where timedate >=1999.01.02);
```

```
KerF> twoparcel: read_parceled_from_path("/tmp/p");
```

Parceled tables automatically have a virtual “date” column.

```
KerF> select avg(vals) from twoparcel where date=1999.01.01
```

vals
1.0

Parceled tables are distinct from standard tables, and this is reflected in the Kerf repl.

```
KerF> kerf_type_name twoparcel  
"parceled table"
```

Since the parceling logically divides by day, there is an implied grouping by day.

```
KerF> select min(vals),min(timedate) from twoparcel
```

vals	timedate
0	1999.01.01
3	1999.01.02

7.5 Network I/O (IPC)

Kerf has a specialized remote procedure call system built on TCP which permits distributing tasks across multiple processes or machines. This is sometimes described as Inter-Process Communication (IPC). To spawn a Kerf process which listens for IPC calls, invoke `kerf` with the `-p` command-line argument and specify a TCP port number:

```
> ./kerf -p 1234
Kerf Server listening on port: 1234
KeRF>
```

The function `open_socket(host, port)` opens a connection to a remote Kerf instance at hostname `host` and listening on port and returns a connection handle. Both `host` and `port` must be strings. Once opened, a connection handle may be closed via `close_socket(handle)`.

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

The function `send_async(handle, x)` will send a string `x` to a remote Kerf instance without waiting for a reply. `x` will be `eval`ed on the remote server, returning 1 on a successful send.

The function `send_sync(handle, y)` will 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_async(c, "foo: 2+3")
1
KeRF> foo
{}
KeRF> send_sync(c, "[foo, foo]")
[5, 5]
KeRF> send_sync(c, "sum range 1000")
499500
```

Network IPC is automatically compressed if Kerf determines that will reduce transfer times. Compression and decompression takes places transparently on each end, so the user typically need not think about it. Kerf will not attempt to recompress already zipped items. It will also not attempt to zip small messages that would not benefit from compression, such as single atoms.

During IPC execution, the constant `.Net.client` contains the current client's unique handle:

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

If defined, an IPC server will call the single-argument function `.Net.on_close` 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
```

Data can be sent to the foreign process using `send_sync` or `send_async` when using these verbs with three arguments. Kerf assumes that the third argument is a list, so if a higher order construct is sent over, it must be payloaded as a list. Arguments are retrieved on the remote side with `$1`.

```
KeRF> c: open_socket("localhost", "1234")
4
KeRF> send_async(c, "foolist: $1",[range(10)])
1
KeRF> send_sync(c,"tbl:$1",[a:[1, 2 ,3],b:["a", "b", "c"]])
```

a	b
1	a
2	b
3	c

7.5.1 Starting an HTTP Server

Starting Kerf with the command-line argument `-P 8080` will start Kerf with an HTTP server opened on the provided port, in this case 8080. The global variable `.Net.parse_request` is pre-populated with a browser-compatible HTML response. This variable is a one-argument function and may be changed as desired.

7.5.2 Backgrounding a Kerf Server

We recommend the UNIX command `screen` for backgrounding Kerf servers. It's possible to use `nohup` or some related form of daemonization, but this has the downside of closing `stdin`, which means that you cannot recover the ability to interact with Kerf via the terminal. The `screen` utility avoids this downside. The process will run in the background, even if you disconnect, and later on you can still type in the console.

Some helpful shell commands:

```
#Try screen instead of nohup

#Installing screen
#if necessary, choose one
sudo yum -y install screen
sudo apt-get install screen
brew install screen
#and so on...

#start a screen session in the background with a Kerf server on port 1234
screen -dm ./kerf -p 1234

#Optional: bring the screen to foreground/terminal
screen -dRR

#Test with another client using the attached test script.
./kerf test-client.kerf

#This has the benefit of retaining STDIN to the server and the server output. (nohup kills stdin.)
```

test-client.kerf

```
//open server: ./kerf -p 1234
//      client: ./kerf test-client.kerf

s: open_socket("localhost","1234")
display send_async(s,"a:$1+1", [range 999])
display send_sync(s, "a:$1", [range 1e4])
```

8 Foreign Function Interface

Kerf provides a Foreign Function Interface (FFI) which makes it possible to call out to C libraries from Kerf and call into Kerf from C. The FFI permits users to supplement Kerf with new IO capabilities, make use of pre-existing libraries and fine-tune high performance applications. In this section we will discuss how to use this facility and describe the data structures and the native API exposed by the Kerf executable.

8.1 C from Kerf

To compile a dynamic library for use with Kerf, you will need `kerf_api.h`, which should be included with the Kerf binary and other documentation. This contains the signatures of exposed `kerf_api_` functions you can use in building dynamic libraries as well as a number of useful constants. If your C source code is stored in `example.c` and this header is in the current directory, compile it from the command line as follows:

For OSX:

```
>cc -m64 -flat_namespace -undefined suppress -dynamiclib example.c -o example.dylib
```

The `-m64` option requests code be generated for a 64-bit architecture. The `-flat_namespace` and `-undefined suppress` are necessary to create a dynamic library which can be linked against the Kerf binary at runtime—this is how your dynamic library can access built-in utility methods from the interpreter itself. The `-o` option specifies a name for the output file, which we suffix with `.dylib` by convention.

For Linux:

```
>cc -m64 -shared -fpic example.c -o example.dylib
```

Linux also requires `-m64`, but by contrast uses `-fpic` and `-shared` to build a shared library which can be dynamically linked to Kerf at runtime. While Linux generally uses `.so` rather than `.dylib` suffixes for shared objects, we will stick to the OSX naming convention for these examples for the sake of consistency.

With this dynamic library compiled, the `dlopen` built-in function can be used to load the library and create a wrapper function for a specific function from the library which can then be invoked in Kerf as if it were an ordinary Kerf function. The Kerf FFI does not marshal Kerf's native data representations into C types—if another representation is desired you must close the gap in your code. This approach simplifies the FFI implementation and permits very efficient data exchange, but requires a great deal of care and attention from library authors.

All values in Kerf are internally stored in KERF structures. This struct is what is called a *tagged union*- some fields have different interpretations dependent on the value of a *tag field*. In our case, the typecode field `t` permits this discrimination.

```
typedef struct kerf0 {
    char m;      // log_2 of memory slab size (internal use only)
    char a;      // attribute flags
    char h;      // log_2 of header subslice size (internal use only)
    char t;      // typecode, as in kerf_type
    int32_t r;   // reference count

    union {
        int64_t i; // value of a Kerf integer or stamp
        double f; // value of a Kerf float
        char c; // value of a Kerf char
        char* s;
        struct kerf0* k;

        struct {
            int64_t n; // in a vector or list, the number of elements
            char g[]; // in a vector or list, the elements themselves
        };
    };
} *KERF, KERFO;
```

Some of the fields of a KERF structure are for internal use by the interpreter and should not be manipulated directly in well-behaved dynamic libraries. The next section will describe the internal representation of various Kerf types in terms of KERF structures. Three fields are of general interest:

- The `r` field contains a *reference count* for the object. Examining this field explicitly is sometimes useful for debugging memory corruption problems. Your code should treat it as read-only.
- The `t` field indicates the object's *typecode*, identical to the typecodes given by `kerf_type`. Negative values indicate vector types.
- The `a` field contains *attribute flag* bits which provide the interpreter with metadata that alters the behavior of values or permits runtime optimizations.

Functions called from Kerf will always take some number of KERF structures as arguments and return a KERF structure. In the event that your function doesn't produce a meaningful return value, return 0- Kerf will coerce this into the equivalent of null.

8.2 Kerf from C

It is easy to call Kerf methods from C. The Kerf binary is compiled using `-rdynamic`, which allows other binaries to load it directly as a shared library. This makes the regular Kerf binary similar to a `.so` shared object file. All of Kerf's symbols are exposed in this manner, although you should limit yourself to the methods beginning with `kerf_api_` as described in `kerf_api.h`. The private Kerf symbols may change at any time. The methods exposed via the API are wrapped and guaranteed to be future-proof.

When calling Kerf from C, remember to call `kerf_api_init` to initialize the Kerf process if it has not already had a chance to initialize itself. Failure to do so may result in undefined behavior.

The following C source file `kerf_from_c.c` illustrates how to call the plain Kerf binary from a C executable. See the comments for compilation instructions. If you prefer, you can load each function individually by using `dlsym`, avoiding the need to suppress unresolved symbols as we do here for the sake of brevity:

```
#include <dlfcn.h>
#include "kerf_api.h"

//OSX:  cc -m64 -rdynamic -flat_namespace -undefined suppress kerf_from_c.c -o kerf_from_c
//LINUX: cc -m64 -rdynamic -fPIC -Wl,--unresolved-symbols=ignore-in-object-files
//      kerf_from_c.c -o kerf_from_c -ldl

int main() {
    void *lib = dlopen("/full/path/to/kerf/binary", RTLD_NOW|RTLD_GLOBAL);
    if(!lib) printf ("A dynamic linking error occurred: (%s)\n", dlerror());

    kerf_api_init(); // always call this first!

    KERF s = kerf_api_new_int(33);
    kerf_api_show(s);
    kerf_api_release(s);

    KERF c = kerf_api_new_charvec("1+1");
    kerf_api_show(c);

    KERF k = kerf_api_interpret(c);
    kerf_api_show(k);
    kerf_api_release(k);

    return 0;
}
```

8.2.1 Kerf `.a` archive and `.o` object files

Depending on your arrangement, you may be granted access to Kerf object files. These `.a` and `.o` files will similarly expose the Kerf methods, with the exception that they may be statically compiled into other binaries. In most cases this is unnecessary, as the regular `-rdynamic` Kerf binary will suffice.

8.3 Reference Counting

Kerf manages dynamically allocated memory using a strategy called *reference counting*. Each KERF object contains a counter of how many references to the object exist throughout the runtime. Whenever an object is used or stored, its reference count is incremented. When an object is no longer needed (perhaps a function using the object completes and returns) this count is decremented. When an object has a reference count of 0, the Kerf runtime knows that it is free to deallocate the object and reclaim the associated memory.

Reference counting is contrasted with *garbage collection*, another popular method of managing dynamic memory. A garbage collector actively *searches* for references to objects and automatically frees objects when they are no longer reachable. Garbage collectors remove the need to explicitly maintain reference counts, but generally incur runtime overhead for performing their heap scanning. Kerf uses reference counting because it has low runtime overhead and, by giving programmers more explicit awareness of memory management, permits more predictable runtime performance.

If the reference counts on an object are not maintained properly, a variety of bugs can occur. If references are not *released* when an object is no longer needed, memory will *leak*- it is no longer useful but cannot be reclaimed for other purposes. In long-running processes this can eventually exhaust system memory. If references are not *retained* when an object is still needed, they can be deallocated out from under your code, allowing data to be overwritten and creating a variety of potential intermittent failures.

Some objects require special consideration. For efficiency purposes, Kerf will sometimes allocate objects which can be passed as arguments to your dynamic libraries on the *stack* instead of its reference-counted *heap*. Stack-allocated objects typically have a reference count of -1 and cannot be retained indefinitely without copying. These and other special cases mean that simply decrementing or incrementing the *r* field of a KERF object is not sufficient for controlling reference counts correctly. The Kerf native API exposes two functions- **kerf_api_release** and **kerf_api_retain**- for this purpose which perform all the necessary bookkeeping. Do *not* directly modify the *r* field of KERF objects!

8.3.1 Releasing

As a general rule, if a KERF object is allocated in a function (as with any of the **kerf_api_new_** functions) it must either be returned from that function or released with **kerf_api_release** before the function returns. The following routine leaks a KERF_CHARVEC. Can you see why?

```
KERF leaky_demo(KERF monad) {
    return kerf_api_call_monad(monad, kerf_api_new_charvec("Hello!"));
}
```

A corrected version would be as follows:

```
KERF leaky_demo(KERF monad) {
    KERF string = kerf_api_new_charvec("Hello!");
    KERF result = kerf_api_call_monad(monad, string);
    kerf_api_release(string);
    return result;
}
```

Any objects which were allocated inside **kerf_api_call_monad** are the responsibility of that routine, and the result it returns can be freely returned from our routine. Objects passed *into* our routine are the responsibility of the code which called our routine, and will be automatically deallocated unless we explicitly retain them. This idea of *local responsibility* is the key to understanding reference counting.

8.3.2 Retaining

In many situations, simply releasing objects you allocate is sufficient, but in some situations we need to explicitly retain an object to ensure it is not deallocated too soon. Consider this pass-through function which prints information about reference counts:

```
KERF show_refcount(KERF x) {  
    printf("incoming reference count: %d\n", x->r);  
    KERF result = kerf_api_retain(x);  
    printf("outgoing reference count: %d\n", result->r);  
    return result;  
}
```

The calling function which passed the arguments into our dynamic library function is responsible for discarding those arguments when our function returns. If we want to return a value that was originally passed to us as an argument, we must retain it to counteract this.

```
KerF> refcount: dlopen("example.dylib", "show_refcount", 1);  
KerF> refcount(5)  
incoming reference count: -1  
outgoing reference count: 1  
5  
KerF> refcount([1,2,3])  
incoming reference count: 2  
outgoing reference count: 3  
[1, 2, 3]  
KerF> a: 4 5 6;  
KerF> refcount(a)  
incoming reference count: 3  
outgoing reference count: 4  
[4, 5, 6]
```

In the first example above, the value was stack-allocated, and the call to **kerf_api_retain** copied this value to the heap, returning a pointer to the new object. *Always* use the result of a call to **kerf_api_retain** instead of the original value.

Retaining is also important if you want to stash a value for later use. Here's an example of some dynamic library functions which hang onto state:

```
KERF stashed_value = 0;

KERF state_init(KERF value) {
    stashed_value = kerf_api_retain(value);
    return 0;
}
KERF state_double() {
    KERF result = stashed_value;
    stashed_value = kerf_api_new_int((result->i) * 2);
    return result;
}
KERF state_free(KERF value) {
    kerf_api_release(stashed_value);
    stashed_value = 0;
    return 0;
}
```

We retain the initial value passed in, since it would otherwise be destroyed when the function returns, as in the previous example. In subsequent calls to `state_double`, we can return the stashed value, relinquishing control of it. The new value we construct each time has a reference count of 1 and can also be returned when the time comes. Freeing our stashed state is included for completeness.

```
KerF> init: dlopen("example.dylib", "state_init", 1);
KerF> step: dlopen("example.dylib", "state_double", 0);
KerF> free: dlopen("example.dylib", "state_free", 0);
KerF> init(5);
KerF> step()
5
KerF> step()
10
KerF> step()
20
KerF> free();
```

8.4 Internal Representations

To work with KERF structures, it is necessary to understand certain details of how they are represented. The implementation of some data structures, however, is kept intentionally opaque. Even dynamic libraries should not depend on distinguishing the internal structure of an enumeration or an index from a mixed-type list, for example. For low-level operations, KERF structures can be directly consulted, and for operations on higher-level datatypes the native API exposes helper functions.

8.4.1 Integers

Kerf Integers have a typecode of `KERF_INT`, and store their values in the `i` field of a KERF structure as a signed 64-bit integer.

8.4.2 Floats

Kerf Floats have a typecode of `KERF_FLOAT`, and store their values in the `f` field of a KERF structure as an IEEE-754 double-precision floating point number.

8.4.3 Characters

Kerf Characters have a typecode of `KERF_CHAR`, and store their values in the `c` field of a KERF structure as UTF-8 characters.

8.4.4 Stamps

Kerf Timestamps have a typecode of `KERF_STAMP`, and store their values in the `i` field of a KERF structure as a signed 64-bit count of nanoseconds since Unix Epoch.

8.4.5 Vectors

Vectors store their data in the `g` field of a KERF structure as a raw, packed array, and use the `n` field to indicate the number of elements they contain. Vectors are typically slab-allocated, and may contain extra allocated space beyond the item count given in `n`, but do not assume this will be the case. See `kerf_api_new_kerf` for an example of how to create a vector from scratch.

8.5 Attribute Flags

The `a` field of a KERF structure is an integer which represents a vector of single-bit *flag* values. Each flag attached to a value represents a boolean piece of metadata which describes properties or configuration settings of the value. Kerf uses many attribute flags for internal bookkeeping, but a few of these flags represent information that could be generally useful for your dynamic libraries.

To test for the presence of a flag, bitwise AND the flag constant with the `a` field:

```
int flag_is_set = ((x->a) & KERF_ATTR_Y) != 0;
```

To set a flag, bitwise OR the flag constant into the `a` field:

```
(x->a) |= KERF_ATTR_Y;
```

To clear a flag, bitwise AND the `a` field with the bitwise complement of the flag constant. Think of it as “keep everything *except* this flag”:

```
(x->a) &= ~KERF_ATTR_Y;
```

The `KERF_ATTR_SORTED` flag indicates that the elements of a vector or list is sorted in ascending order. The same information can be queried in Kerf using the **sort.debug** built-in function.

```
KERF is_sorted(KERF list) {  
    return kerf_api_new_int((list->a & KERF_ATTR_SORTED) != 0);  
}
```

```
KerF> f: dload("example.dylib", "is_sorted", 1);  
KerF> f(1 2 3)  
1  
KerF> f(3 1 2)  
0
```

The `KERF_ATTR_BYTES` flag applies specifically to character vectors. If set, their contents will be pretty-printed as a series of hexadecimal bytes with a `0x` prefix instead of as characters.

```
KERF demo_attr_bytes_set(KERF charvec) {  
    charvec = kerf_api_retain(charvec); // returning a modified argument, must retain  
    charvec->a |= KERF_ATTR_BYTES;  
    return charvec;  
}
```

```
KerF> dload("example.dylib", "is_sorted", 1)("ABCDE")  
0x4142434445
```

The KERF_ATTR_DISK flag indicates whether a value is located in disk-backed storage, and should be considered a read-only attribute. The same information is reflected in the `is_disk` column of the results from `meta_table`.

```
KERF demo_attr_disk_get(KERF x) {  
    return kerf_api_new_int((list->a & KERF_ATTR_DISK) != 0);  
}
```

```
KeRF> b: create_table_from_csv("breakfast.bin", "breakfast.csv", "SI", 1)
```

Ingredient	Quantity
Bacon	5
Egg	2
English Muffin	1
Bearnaise Sauce	1

```
KeRF> f: dlopen("example.dylib", "demo_attr_disk_get", 1);
```

```
KeRF> f b
```

```
1
```

```
KeRF> f {{foo: 1 2}}
```

```
0
```

8.6 Native API

The signatures of the following methods are described in a machine-readable form in `kerf_api.h`.

8.6.1 `kerf_api_append` - Append to List

KERF `kerf_api_append(KERF x, KERF y)`

Append an object `y` to the list `x`. If the original list can be referenced from elsewhere (if it is stored in a global, for example), you must retain it before appending. It is safe to call `kerf_api_append` on a list you construct yourself without any special work.

```
KERF append_demo(KERF base) {
    KERF tail = kerf_api_new_int(54);
    KERF list = kerf_api_append(kerf_api_retain(base), tail);
    kerf_api_release(tail);
    return list;
}
```

```
KerF> f: dlopen("example.dylib", "append_demo", 1);
KerF> f([])
[54]
KerF> f(1 2 3)
[1, 2, 3, 54]
KerF> f(5)
[5, 54]
KerF> a: 4 5 6;
KerF> f(a)
[4, 5, 6, 54]
KerF> a
[4, 5, 6]
```

Note that the value stored in the global `a` *was not modified*- we altered a copy.

8.6.2 `kerf_api_call_dyad` - Call Dyad

`kerf_api_call_dyad(KERF func, KERF x, KERF y)`

Call a dyadic (binary) function and return the result.

```
KERF dyad_demo(KERF dyad) {
    KERF a1 = kerf_api_new_int(3);
    KERF a2 = kerf_api_new_int(5);
    KERF ret = kerf_api_call_dyad(dyad, a1, a2);
    kerf_api_release(a1);
    kerf_api_release(a2);
    return ret;
}
```

```
KerF> dlopen("example.dylib", "dyad_demo", 1)({[x,y] [x,y,x+y,x*y]})
[3, 5, 8, 15]
```

8.6.3 `kerf_api_call_monad` - Call Monad

`kerf_api_call_monad(KERF func, KERF x)`

Call a monadic (unary) function and return the result.

```
KERF monad_demo(KERF monad) {
    KERF arg = kerf_api_new_int(17);
    KERF ret = kerf_api_call_monad(monad, arg);
    kerf_api_release(arg);
    return ret;
}
```

```
KerF> dload("example.dylib", "monad_demo", 1)({[x] [x,0,2*x,x*x]})
[17, 0, 34, 289]
```

8.6.4 `kerf_api_call_nilad` - Call Nilad

`kerf_api_call_nilad(KERF func)`

Call a niladic function (one which takes no arguments) and return the result.

```
KERF nilad_demo(KERF nilad) {
    KERF temp = kerf_api_call_nilad(nilad);
    kerf_api_show(temp);
    kerf_api_release(temp);
    return kerf_api_call_nilad(nilad);
}
```

```
KerF> a:[25]
[25]
KerF> dload("example.dylib", "nilad_demo", 1)({[] a[0]++; [3,2,a[0]]})
[3, 2, 26]    // printed by kerf_api_show
[3, 2, 27]    // returned
```

8.6.5 `kerf_api_copy_on_write` - Copy On Write

`KERF kerf_api_copy_on_write(KERF x)`

If `x` is referenced by multiple owners or stack-allocated, make a shallow copy and return this copy. Otherwise, it is safe to modify `x` in place. **`kerf_api_copy_on_write`** may free the object passed in, so if you don't want the original destroyed it must be retained first.

8.6.6 kerf_api_get - Kerf Get

KERF kerf_api_get(KERF x, KERF index)

Index into the KERF list x. Equivalent to the normal Kerf expression x[index].

```
KERF get_demo(KERF list, KERF index) {
    KERF zero = kerf_api_new_int(0);
    KERF first = kerf_api_get(list, zero);
    kerf_api_show(first);
    kerf_api_release(zero);
    kerf_api_release(first);
    return kerf_api_get(list, index);
}
```

```
KerF> f: dlopen("example.dylib", "get_demo", 2);
KerF> f("ABCD", 2)
`"A"           // printed by kerf_api_show
  `"C"         // returned
KerF> f(1 3 7 10, 2 0 2)
1              // printed by kerf_api_show
[7, 1, 7]     // returned
```

8.6.7 kerf_api_init - Initialize the Kerf Process

int kerf_api_init()

Call this method at the beginning of any API interaction where Kerf is not the initiating process. This will ensure Kerf is initialized. Loading Kerf as a shared object from C (or any other language) will require **kerf_api_init**. If Kerf launches on its own, then it will call this function internally, but it is still good practice to include the initialization in code loaded as dynamic libraries. There is no harm in calling the initialization multiple times, beyond wasted effort.

8.6.8 kerf_api_interpret - Interpret String

KERF kerf_api_interpret(KERF charvec)

Execute a KERF_CHARVEC (string) as if via the function **eval**.

```
KERF interpret_demo() {
    KERF s1 = kerf_api_new_charvec("a: 1+2");
    KERF s2 = kerf_api_new_charvec("2*range a");
    KERF r1 = kerf_api_interpret(s1);
    KERF r2 = kerf_api_interpret(s2);
    kerf_api_release(s1);
    kerf_api_release(s2);
    kerf_api_show(r1);
    kerf_api_show(r2);
    kerf_api_release(r1);
    kerf_api_release(r2);
    return 0;
}
```

```
KeRF> dlopen("example.dylib", "interpret_demo", 0)()
3
[0, 2, 4]
```

8.6.9 kerf_api_len - Kerf Length

int64_t kerf_api_len(KERF x)

Determine the length of a KERF object x, as given by the function **len**. This routine is more general than consulting the n field of a KERF structure, and will produce sensible results for non-vector types.

```
KERF len_demo() {
    KERF string = kerf_api_new_charvec("Hello, C!");
    KERF number = kerf_api_new_int(43);
    KERF list = kerf_api_new_list();
    printf("%d %d %d\n",
        (int)kerf_api_len(string),
        (int)kerf_api_len(number),
        (int)kerf_api_len(list)
    );
    kerf_api_release(string);
    kerf_api_release(number);
    kerf_api_release(list);
    return 0;
}
```

```
KeRF> dlopen("example.dylib", "len_demo", 0)();
9 1 0
```

8.6.10 `kerf_api_new_charvec` - New Kerf String

KERF `kerf_api_new_charvec(char* cstring)`

A wrapper for `kerf_api_new_kerf` which allocates and initializes a `KERF_CHARVEC` (string) from a null-terminated C string. The supplied string will be copied into the new vector; subsequent mutations will not propagate from one to the other.

8.6.11 `kerf_api_new_float` - New Kerf Float

KERF `kerf_api_new_float(double n)`

A wrapper for `kerf_api_new_kerf` which allocates and initializes a single Kerf Float from an IEEE-754 double-precision floating point number `n`.

8.6.12 `kerf_api_new_int` - New Kerf Integer

KERF `kerf_api_new_int(int64_t n)`

A wrapper for `kerf_api_new_kerf` which allocates and initializes a single Kerf Integer from a signed 64-bit C integer `n`.

8.6.13 `kerf_api_new_kerf` - New Kerf Object

KERF `kerf_api_new_kerf(char type, int64_t length)`

Allocate a raw KERF structure with a given type. For scalar types like `KERF_INT`, `length` should be 0. For vector types, `length` will be the number of entries the object will contain. This routine allocates space in Kerf's internal memory pool and initializes reference counts appropriately.

```
KERF intvec_demo(KERF base, KERF count) {
    KERF ret = kerf_api_new_kerf(KERF_INTVEC, count->i);
    for(int x = 0; x < count->i; x++) {
        ((int64_t*)(ret->g))[x] = (base->i) + x;
    }
    return ret;
}
```

```
KeRF> f: dload("example.dylib", "intvec_demo", 2);
KeRF> v: f(70000, 5)
      [70000, 70001, 70002, 70003, 70004]
KeRF> kerf_type_name v
      "integer vector"
```

8.6.14 `kerf_api_new_list` - New Kerf List

KERF `kerf_api_new_list()`

Allocate an empty list. Useful in combination with `kerf_api_append`.

```
KERF list_demo() {  
    return kerf_api_new_list();  
}
```

```
KerF> v: dload("example.dylib", "list_demo", 0)()  
[]  
KerF> kerf_type v  
6  
KerF> kerf_type_name v  
"list"  
KerF> len v  
0
```

8.6.15 `kerf_api_new_map` - New Kerf Map

KERF `kerf_api_new_map()`

Allocate an empty map.

```
KERF map_demo() {  
    return kerf_api_new_map();  
}
```

```
KerF> v: dload("example.dylib", "map_demo", 0)()  
  
KerF> kerf_type v  
7  
KerF> kerf_type_name v  
"map"  
KerF> len v  
1
```


8.6.16 `kerf_api_new_stamp` - New Kerf Timestamp

KERF `kerf_api_new_stamp(int64_t nanoseconds)`

A wrapper for `kerf_api_new_kerf()` which allocates and initializes a single Kerf Stamp from a signed 64-bit count of nanoseconds since Unix Epoch n.

```
KERF stamp_demo(KERF ns) {
    KERF epoch = kerf_api_new_stamp(0);
    kerf_api_show(epoch);
    kerf_api_release(epoch);
    return kerf_api_new_stamp(ns->i);
}
```

```
KerF> t: dload("example.dylib", "stamp_demo", 1)(3000000)
00:00:00.000    // printed by kerf_api_show
    00:00:00.003 // returned
KerF> t[["year", "month", "day"]]
[1970, 1, 1]
```

8.6.17 `kerf_api_nil` - Kerf Nil

KERF `kerf_api_nil()`

Allocate an object representing nil/null.

```
KERF nil_demo() {
    return kerf_api_nil();
}
```

```
KerF> v: dload("example.dylib", "nil_demo", 0)();

KerF> kerf_type v
5
KerF> kerf_type_name v
"null"
KerF> len v
1
```

8.6.18 `kerf_api_release` - Release Kerf Reference

void `kerf_api_release(KERF x)`

Reduce the reference count for a KERF object. Any objects which are allocated in a dynamic library call and not returned or otherwise stored must be manually released, or they will not be reclaimed.

8.6.19 `kerf_api_retain` - Retain Kerf Reference

KERF `kerf_api_retain(KERF x)`

Increase the reference count for a KERF object. This process may require making a copy of the source object.

8.6.20 `kerf_api_set` - Kerf Set

KERF `kerf_api_set(KERF x, KERF index, KERF replacement)`

Index into the KERF list `x` and replace the element with `replacement`, returning the modified list. Similar to the normal Kerf expression `x[index]:replacement`, but incapable of spread assignment. This operation will modify `x` in place.

```
KERF set_demo(KERF list, KERF index, KERF rep) {  
    return kerf_api_set(kerf_api_retain(list), index, rep);  
}
```

```
KeRF> f: dload("example.dylib", "set_demo", 3);  
KeRF> f(1 2 3, 1, 99)  
[1, 99, 3]
```

8.6.21 `kerf_api_show` - Show Kerf Object

KERF `kerf_api_show(KERF x)`

Print a prettyprinted representation of a KERF structure `x` to `stdout`, as displayed by the REPL, and return the structure unchanged. This routine primarily exists for debugging purposes.

```
KERF show_demo(KERF argument) {  
    kerf_api_show(argument);  
    KERF string = kerf_api_new_charvec("Hello, C!");  
    kerf_api_show(string);  
    kerf_api_release(string);  
    return 0;  
}
```

```
KeRF> dload("example.dylib", "show_demo", 1)(1 2 3);  
[1, 2, 3]  
"Hello, C!"
```

8.7 The Kerf IPC Protocol (KIP)

Another way for other languages to interoperate with Kerf is networked IPC. In this section we will provide a description of this protocol sufficient for writing connectors in another language of your choice. Example libraries in C, Python and Java are available on the Kerf website or by request, and **Kerf IPC with Python** provides a step-by-step explanation of building such a connector. See **Network I/O** for additional information.

The Kerf IPC protocol (referred to hereafter as KIP for the sake of brevity) uses TCP as its transport layer. Individual messages consist of a *header* followed by a *payload*. The header contains information about the type of message and its size. The payload can take several forms, depending on message type. The entire message is 0-padded to an even power of two, which helps avoid memory fragmentation in low-level implementations of the protocol and makes a raw `memcpy()` of the message safe.

To execute code on a remote Kerf instance it must be started with the `-p` flag. Your application can then open a TCP connection, transmit a message and wait for a response. Note that if the TCP connection is closed before the response has been transmitted, any in-progress execution on the remote Kerf instance will be aborted. After you receive a response you may send another message or close the connection.

The binary representation of serialized Kerf data structures is beyond the scope of this document and highly subject to change in future revisions of the language, but it is also possible to send a payload as a UTF-8 encoded JSON string. We will describe this mode of operation in detail. Note that when transmitting JSON data a message is restricted to roughly 4 gigabytes, but this limitation is not present for other message types. If transmission of bulk data is a limiting factor in your application, consider using a pair of communicating Kerf processes rather than a custom connector.

The structure and content of the header may change slightly in future versions of Kerf, so this document will focus only on the most essential fields.

8 bits	8 bits	8 bits	8 bits
0x00	0x00	0x00	0x00
Execution Type	Response Type	Display Type	0x00
0x00	0x00	0x00	0x00
Wire Size			
Shard Size	0x00	0x00	0xFF
0x01	0x00	0x00	0x00
Payload Size			
0x00	0x00	0x00	0x00

KIP Header Structure

8.7.1 Execution Type

An 8-bit signed integer indicating how a message should be executed by the recipient. For our purposes, this will always be 4, which indicates the payload is a JSON-encoded string.

8.7.2 Response Type

An 8-bit signed integer indicating the way the response should be formatted. 0 indicates no response is desired, 1 requests the complete JSON-encoded response.

8.7.3 Display Type

An 8-bit signed integer indicating how the Kerf process should display incoming messages, if at all. 0 indicates nothing should be printed when a message comes in, 1 indicates the incoming message content should be printed, 2 indicates the result of executing a message should be printed and 3 indicates that both should be printed. 3 is useful for debugging, but 0 or 1 may be most desirable in real applications.

8.7.4 Wire Size

A 32-bit unsigned integer in network byte order (big-endian) indicating the size, in bytes, of the payload plus 16. Note that this value is precisely the same as `pow(2, shard size)`.

8.7.5 Shard Size

An 8-bit signed integer indicating the \log_2 of the length of the payload in bytes plus 16.

8.7.6 Payload Size

A 32-bit unsigned integer in native byte order (little-endian) indicating the size, in bytes, of the payload. The wire size will always exceed this size, and after the header and payload the remainder of a message will be 0-padded.

9 Built-In Function Reference

9.1 **abs** - Absolute Value

`abs(x)`

Calculate the absolute value of x. Atomic.

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

9.2 **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]
```

9.3 **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]
```

9.4 **and** - Logical AND

`and(x, y)`

Calculate the logical *AND* of x and y. This operation is equivalent to the 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]
```

9.5 `append_table_from_csv` - Append Table From CSV File

```
append_table_from_csv(tableFile, csvFile, fields, n)
```

Equivalent to `create_table_from_csv` which appends to any pre-existing file instead of overwriting.

9.6 `append_table_from_fixed_file` - Append Table From Fixed-Width File

```
append_table_from_fixed_file(tableFile, fixedFile, attributes)
```

Equivalent to `create_table_from_fixed_file` which appends to any pre-existing file instead of overwriting.

9.7 `append_table_from_psv` - Append Table From PSV File

```
append_table_from_psv(tableFile, psvFile, fields, n)
```

Equivalent to `create_table_from_psv` which appends to any pre-existing file instead of overwriting.

9.8 `append_table_from_tsv` - Append Table From TSV File

```
append_table_from_tsv(tableFile, tsvFile, fields, n)
```

Equivalent to `create_table_from_tsv` which appends to any pre-existing file instead of overwriting.

9.9 `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]
```

9.10 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]
```

9.11 asof_join - Asof Join

asof_join(x, y, k1, k2)

Perform a “fuzzy” **left_join**. See **Joins**.

9.12 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]
```

9.13 atlas - Atlas Of

atlas(map)

Create an atlas from a map or list of maps. An atlas is the schemaless NoSQL equivalent of a table. This gives tables in Kerf the option of being unstructured, or schema-less, for dealing with highly irregular data. Atlases are automatically indexed in such a way that all key-queries are indexed. Atlases allow for NoSQL queries.

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

9.14 atom - Is Atom?

atom(x)

A predicate which returns 0 if x is a list or vector, 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
```

9.15 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
```


9.16 bars - Time Bars, Sample Buckets, etc.

`bars(x, y)`

Round the elements of a list `y` to multiples of `x`. Particularly useful in the context of “time bucketing” sample data. Equivalent to `x * floor y/x`.

```
KeRF> bars(2, [1, 1.2, 1.4, 2, 2.5, 3, 3.1, 5])
[0, 0, 0, 2, 2, 2, 2, 4]
KeRF> t: [1:02, 1:03, 1:14, 1:15, 1:35];
KeRF> bars(15, t['minute']) //roll up times into 15 minute bars
[0, 0, 0, 15, 30]
```

The `bars` function is also defined for simple relative date-times when used on timestamps for generating bars on tables that are across time boundaries. This is generally faster and more convenient than the equivalent multiple “group by” statement on multiple clocks.

```
KeRF> ts: {{value:range(10),time: 1999.01.01 + 1i * mapright range(10)}}
```

value	time
0	1999.01.01
1	1999.01.01T00:01:00.000
2	1999.01.01T00:02:00.000
3	1999.01.01T00:03:00.000
4	1999.01.01T00:04:00.000
5	1999.01.01T00:05:00.000

```
KeRF> select avg(value) from ts group by bars(2i,time) //select 2 minute bars
```

time	value
1999.01.01	0.5
1999.01.01T00:02:00.000	2.5
1999.01.01T00:04:00.000	4.5

When using `bars` with relative date-times, it may be useful to define a starting point as an optional third argument.

```
KeRF> select avg(value) from ts group by bars(2i,time,1999.01.01T00:01:00)
```

time	value
1998.12.31T23:59:00.000	0.0
1999.01.01T00:01:00.000	1.5
1999.01.01T00:03:00.000	3.5
1999.01.01T00:05:00.000	5.5
1999.01.01T00:07:00.000	7.5
1999.01.01T00:09:00.000	9.0

9.17 **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
```

9.18 **btree** - BTree

`btree(x)`

Equivalent to **indexed**.

9.19 **bucketed** - Bucket Values

`bucketed(x, y)`

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

9.20 **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**. **car** is a reference to the Lisp primitive of the same name, which selected the first element of a pair. See **cdr**.

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

9.21 **cdr** - Contents of Decrement 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. See **car**.

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

9.22 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!"
```

9.23 char - Cast to Char

char(x)

Cast a number or list x to a char or string, respectively. To reverse this operation, use **int**.

```
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!"
```

9.24 checksum - Object Hashcode

checksum(x)

Produce an integer hashcode for any Kerf object x. The precise algorithm used by checksum is not specified and may change, but results will be consistent across runs and any two objects for which **match** produces 1 will have the same hashcode.

```
KeRF> checksum 1 2 3
-7744665892335545863
KeRF> checksum 5
1200461294887951755
```

9.25 close_socket - Close Socket

close_socket(handle)

Given a socket handle as obtained with **open_socket**, close the connection. See **Network I/O**.

9.26 combinations - Combinations

```
combinations(x, n)
combinations(x, n, repeats)
```

Produce a list of all the distinct subsets of `x` which contain `n` elements. Normally this will operate on the unique elements of `x`, but if `repeats` is truthy all elements will be preserved:

```
KeRF> combinations(6 7 8, 2)
[[6, 7],
 [6, 8],
 [7, 8]]
KeRF> combinations(3 5 5 7, 3)
[[3, 5, 7]]
KeRF> combinations(3 5 5 7, 3, 1)
[[3, 5, 5],
 [3, 5, 7],
 [3, 5, 7],
 [5, 5, 7]]
```

If `x` is a map, operate on its keys:

```
KeRF> combinations({foo: 1, bar: 2, quux: 3}, 2)
[["foo", "bar"],
 ["foo", "quux"],
 ["bar", "quux"]]
```

9.27 compressed - Compressed Vector Of

```
compressed(x)
compressed(x, type)
```

Create a *zip*, or compressed vector, from a vector `x`. Compressed vectors consume less memory at runtime, and if serialized to disk will consume less disk space. Precise savings will be data-dependent. Otherwise, they may be treated like an ordinary vector. Note that it is invalid to apply **compressed** to a non-vectorizable object.

```
KeRF> compressed 1 2 3
COMPRESSED[1, 2, 3]
KeRF> compressed [12:00, 11:58]
COMPRESSED[12:00:00.000, 11:58:00.000]
```

If **compressed** is provided with a second character argument, `type`, you can specify a variety of specialized compressed vector types using typecodes as described for **Delimited File IO**. These specialized types can save additional space by reducing the number of bytes used to represent numbers, at the cost of some precision. Presently all compressed vectors use LZ4 compression for its favorable balance of speed and compression ratio. In the future additional algorithms may be available via typecodes which are better suited to specific types of data.

9.28 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]
```

9.29 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]
```

9.30 count - Count

`count(x)`

Equivalent to **len**.

9.31 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
```

9.32 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
```

9.33 `create_table_from_csv` - Create Table From CSV File

```
create_table_from_csv(tableFile, csvFile, fields, n)
```

Load a Comma-Separated Value file and create a table which is serialized on disk. If the table file already exists, this operation will overwrite. `tableFile` and `csvFile` are filenames on disk. `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. See **Striped Files**.

9.34 `create_table_from_fixed_file` - Create Table From Fixed-Width File

```
create_table_from_fixed_file(tableFile, fixedFile, attributes)
```

Load a file with fixed-width columns and create a table which is serialized on disk. If the table file already exists, this operation will overwrite. `tableFile` and `fixedFile` are filenames on disk. `attributes` is a map specifying the format of the fixed-width file, as in `read_table_from_fixed_file`. See **Striped Files**.

9.35 `create_table_from_psv` - Create Table From PSV File

```
create_table_from_psv(tableFile, psvFile, fields, n)
```

Equivalent to `create_table_from_csv` which processes pipe-delimited fields (`|`) instead of comma-delimited fields.

9.36 `create_table_from_tsv` - Create Table From TSV File

```
create_table_from_tsv(tableFile, tsvFile, fields, n)
```

Equivalent to `create_table_from_csv` which processes tab-delimited fields instead of comma-delimited fields.

9.37 cross - Cartesian Product

`cross(x, y)`

Pair up each element of `x` with each element of `y`. If `x` or `y` is a map, use the values:

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

9.38 deal - Deal

`deal(count)`

`deal(count, x)`

Select `count` random elements from the list `x` without repeats.

```
KeRF> deal(5, 21 31 41 51 61 71 81)
[81, 21, 61, 51, 41]
KeRF> deal(4, "ABCD")
"CDBA"
```

If `x` is a map or table, select keys.

```
KeRF> deal(3, {a: 99, b: 22, c: 41, d: 55})
["d", "a", "b"]
KeRF> deal(2, {{a: 99 98, b: 21 22, c: 33 34}})
["c", "b"]
```

If `x` is a number, select from `range(x)`:

```
KeRF> deal(10, 10)
[7, 4, 0, 2, 9, 8, 1, 6, 3, 5]
KeRF> deal(5, 5.0)
[2, 3, 1, 0, 4]
```

When `x` is omitted, the `range` of `count` is used as the list.

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

9.39 delete_keys - Delete Keys

`delete_keys(x, y)`

Remove elements from a list `x` at a specified index or indices `y`. All the supplied indices must fall within the dimensions of the original list.

```
KeRF> "ABCDEF" delete_keys 2
"ABDEF"
KeRF> "ABCDEF" delete_keys 2 4
"ABDF"
```

If `x` is a map or table, remove a set of entries with keys `y`:

```
KeRF> delete_keys({a:4, b:9, c:1}, ["c", "a", "c", "f"])
{b:9}
KeRF> delete_keys({{a:4, b:9, c:1}}, ["c", "a", "c", "f"])
```

b
9

9.40 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]
```


9.41 **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. See **File I/O**.

9.42 **display** - Display

```
display(x)
```

Print a display representation of data to standard output. See **General I/O**.

9.43 **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"
```

9.44 **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]
```

9.45 dload - Dynamic Library Load

`dload(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. The current implementation of **dload** permits a maximum `argcount` of 8.

Let's look at a very simple C function which can be called from Kerf:

```
#include <stdio.h>
#include "kerf_api.h"

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

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

For more information about writing dynamic libraries for use in Kerf, see **FFI**.

9.46 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
```

9.47 drop - Drop Elements

`drop(count, x)`

Remove `count` elements from the beginning of the list `x`. 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"
```

9.48 `emu_debug_mode` - Toggle Bytecode Debugger

`emu_debug_mode(x)`

If `x` is truthy, enable the bytecode debugger. Otherwise, disable it. The bytecode debugger displays information about each bytecode as it is executed by Kerf's inner interpreter. This feature only works on debug builds of Kerf and will otherwise have no effect. The details of the information exposed by this feature are subject to change and beyond the scope of this document.

```
KerF> emu_debug_mode 1
[DEBUG] puts.c:30: =====frame top =====
Stack 7 (t:2 n:1): 1
Stack 6 (t:2 n:0): 0
Stack 5 (t:2 n:0): 0
Stack 4 (t:2 n:0): 0
Stack 3 (t:2 n:0): 0
Stack 2 (t:0 n:72354530896904192): [$1].Net.client: $1
Stack 1 (t:2 n:0): 0
Stack 0 (t:2 n:0): 0
[DEBUG] puts.c:36: =====stack bottom=====
Stack: framept: 8, framebot: 8
Execute instruction: 20
```

9.49 `enlist` - Enlist Element

`enlist(x)`

Wrap any element `x` in a list.

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

9.50 `enum` - Enumeration

`enum(x)`

Equivalent to `hashed`.

9.51 `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`.

```
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"]
```

If `x` is a list, generate the Cartesian Product over the ranges of each element of `x`. This operation is sometimes called *odometer*, for the way the resulting lists resemble the rolling digits of a car's odometer.

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

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

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

9.52 equal - Equal?

`equal(x, y)`

A predicate which returns 1 if `x` is equal to `y`. Equivalent to **equals**. 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, .1000000000000001)
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]
```

9.53 equals - Equals?

`equals(x, y)`

Equivalent to `equal`.

9.54 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]
```

9.55 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]
```

9.56 eval - Evaluate

`eval(x)`

Evaluate a string `x` as a Kerf expression. Atomic down to strings.

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

9.57 except - Except

`except(x, y)`

Remove all the elements of `y` from `x`. Equivalent to `x[which not 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]
```

9.58 **exit** - Exit

```
exit()
exit(code)
```

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

```
KerF> exit(1)
```

9.59 **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]
```

9.60 **explode** - Explode

```
explode(key, x)
```

Violently and suddenly split the list x at instances of key . 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]]
```

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"]
```

9.61 **extract** - Extract From Table

`extract(x)`

Isolate simple vector or scalar values from a table. **extract** is mainly intended for unpacking the results from SELECT queries. If the table has a single row and a single column, **extract** will retrieve a scalar value. A single row and multiple columns naturally becomes a list:

```
KerF> extract SELECT a FROM {{a:55, b:27}}
55
KerF> extract SELECT a,b FROM {{a:55, b:27}}
[55, 27]
```

A single column and multiple rows (or no rows) will be unpacked into a vector:

```
KerF> extract SELECT a FROM {{a:55 67, b:27 99}}
[55, 67]
KerF> extract SELECT a FROM {{a:55 67, b:27 99}} WHERE a=0
INT[]
```

In other cases, **extract** is roughly equivalent to **xvals**:

```
KerF> extract {{a:1 2, b:3 4}}
[[1, 2], [3, 4]]
KerF> xvals {{a:1 2, b:3 4}}
[[1, 2], [3, 4]]
```

9.62 **filter** - Filter

`filter(f, x)`

Apply a predicate `f` to each element of `x`, and return a list of elements where `f` produces a truthy value:

```
KerF> (not isnull) filter [3, null, 27, nil, 39]
[3, 27, 39]
KerF> {[x] x > 100} filter 20 19 130 5 -2 200 119
[130, 200, 119]
KerF> {[x] len(x) = 4} filter ["apple","pear","knife","lock","spider"]
["pear", "lock"]
```

If `x` is a map, **filter** will operate on the values of `x` and produce a map result:

```
KerF> filter(isnull, {a:23,b:null,c:22,e:null})
{b:null, e:null}
KerF> {[x] len(x) > 1} filter {foo:1 2 3, bar:4, quux:5 6}
{foo:[1, 2, 3], quux:[5, 6]}
```

For tables, prefer using SQL syntax and the more convenient `WHERE` clause.

9.63 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**.

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

9.64 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]
```

9.65 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
```


9.66 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]
```

9.67 format - Format String

`format(x, y)`

Convert the elements of a list `y` into a single formatted string based on the string `x`. **format** provides a subset of the functionality of the ubiquitous C `printf` function, with some Kerf-specific convenience features.

A format string contains one or more *format sequences*. Each format sequence corresponds sequentially to one of the elements of `y`. Format sequences begin with the character `%` and end with a *format character* which specifies how to interpret the argument. The special combination `%%` produces a literal `%` character in output, and all characters outside format sequences will likewise be preserved as-is.

Between the `%` and the format character there may optionally be a *width* and/or *precision*, separated by a decimal point. The interpretation of each of these arguments depends on the format character. If the format character were `X`, the following would be valid format sequences:

```
%X      // simple format
%12X     // width only
%.5X     // precision only
%12.5X   // width (12) and precision (5)
```

Character	Behavior
%	Literal character %. This sequence does not consume an argument.
s	Format the argument as a string. Non-string arguments will be converted to strings as by the primitive rep . Results will be padded by preceding spaces if less than <i>width</i> characters long when specified. <i>precision</i> is ignored.
d	Format the argument as a decimal (integer) number. Floating point numbers will be truncated. Results will be padded by preceding spaces if less than <i>width</i> characters long when specified. Numeric results will be padded with preceding zeroes if less than <i>precision</i> digits long when specified.
f	Format the argument as a float. Results will be padded by preceding spaces if less than <i>width</i> characters long when specified. Numeric results will be shown with <i>precision</i> decimal places when specified.

Format Characters

Let's look at a few examples:

```
KeRF> "%s %d %% %f " format ["text", 5, 7.8]
"text 5 % 7.800000"
KeRF> "%s | %s | [%5s]" format ["foo", 1 2 3, "abc"]
"foo | [1, 2, 3] | [ abc]"
KeRF> "[%f] [%.2f] [%6.2f]" format take(3, 1.23456)
"[1.234560] [1.23] [ 1.23]"
KeRF> "[%d] [%6d] [%.8d] [%6.4d]" format take(4, 123)
"[123] [ 123] [00000123] [ 0123]"
```

9.68 format_stamp - Format Timestamp

`format_stamp(x, y)`

Convert a timestamp `y` into a string representation based on the string `x`. Right-atomic. The format string `x` builds on the the capabilities of the standard C `strftime` function, adding support for display of milliseconds or nanoseconds. To turn a formatted string back into a Kerf timestamp, see **parse_stamp**.

As in **format**, a format string contains one or more *format sequences*. Each format sequence corresponds sequentially to one of the elements of `y`. Format sequences begin with the character `%` and end with a *format character* which specifies how to interpret the argument. All characters outside format sequences will be preserved as-is.

```
KeRF> out "%t%D %T.%q" format_stamp now() // full nanosecond accuracy
04/29/16 18:18:10.316715000
KeRF> "Thank god it's %A!" format_stamp now()
"Thank god it's Friday!"
```

Character	Behavior	Example
%	Literal character %.	%
a	Abbreviated weekday name	Fri
A	Full weekday name	Friday
b	Abbreviated month name	Apr
B	Full month name	April
c	Alias for %a %b %d %T %Y	Fri Apr 29 16:23:24 2016
C	Year divided by 100 and truncated to integer (00-99)	20
d	Day of the month, zero-padded (01-31)	29
D	Alias for %m/%d/%y	04/29/16
e	Day of the month, space-padded (1-31)	29
F	Alias for %Y-%m-%d	2016-04-29
g	Week-based year, last two digits (00-99)	16
G	Week-based year	2016
h	Alias for %b	Apr
H	Hour in 24h format (00-23)	16
I	Hour in 12h format (01-12)	04
j	Day of the year (001-366)	120
m	Month as a decimal number (01-12)	04
M	Minute (00-59)	23
n	Literal newline character	
N	Decimal nanoseconds (Kerf-specific)	997456000
p	AM or PM designation	PM
q	Decimal milliseconds (Kerf-specific)	997
Q	Decimal microseconds (Kerf-specific)	997456
r	Alias for %I:%M:%S %p"	04:23:24 PM
R	Alias for %H:%M	16:23
S	Second (00-61)	24
t	Literal tab character	
T	Alias for %H:%M:%S (ISO 8601 time)	16:23:24
u	ISO 8601 weekday as number with Monday as 1 (1-7)	5
U	Week number from Sunday (00-53)	17
V	ISO 8601 week number (00-53)	17
w	Weekday as a decimal number from Sunday (0-6)	5
W	Week number from Monday (00-53)	17
x	Alias for %D	04/29/16
X	Alias for %T	16:23:24
y	Year, last two digits (00-99)	16
Y	Year	2016
z	ISO 8601 offset from UTC in timezone	-0700
Z	Abbreviated timezone name, if any	UTC

Timestamp Format Characters

9.69 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]
```

9.70 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]
```

9.71 has_column - Table Has Column?

`has_column(table, key)`

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

```
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
```

9.72 has_key - Has Key?

`has_key(x, key)`

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

```
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 `key` 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, key)`.

9.73 hash - Hash

`hash(x)`

Equivalent to `hashed`.

9.74 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> shell "wc -c a.data"
[" 1048576 a.data"]
KeRF> shell "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"]
```

9.75 **help** - Help Tool

`help(x)`

Query the global **.Help** table. Calling with an empty string lists the help subjects. Calling with a subject or entry will provide a table describing usage.

9.76 **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
```

9.77 **ifnull** - If Null?

`ifnull(x)`

Equivalent to **isnull**.

9.78 **implode** - Implode

`implode(key, x)`

Violently and suddenly join the elements of the list x intercalated with key. 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]
```

9.79 **in** - In?

`in(key, x)`

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

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

9.80 index - Index

`index(x)`

Equivalent to **indexed**.

9.81 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. Indexed columns in tables can help with performance if the column is time oriented, or a float/int which is often used as a query key. Indexes incur memory overhead, so prefer a **sort** on a primary column.

```
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]
```

9.82 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 character or string, produce character codes. To reverse this operation, use **char**. If you want the numeric equivalent of a string, see **parse_int**.

```
Kerf> int `A
65
Kerf> int "100"
[49, 48, 48]
Kerf> int "Hello, Kerf!"
[72, 101, 108, 108, 111, 44, 32, 75, 101, 114, 102, 33]
```

9.83 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"

```

9.84 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, 0, {a:0}]

```

9.85 join - Join

join(x, y)

Form a list by catenating x and y.

```

KeRF> join(1, 2)
[1, 2]
KeRF> join(2, 3 4)
[2, 3, 4]
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]
KeRF> "foo"#{rep 23}#!"
"foo 23!"

```

9.86 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]}"

```


9.87 kerf_from_json - Convert JSON to Kerf

kerf_from_json(string)

Convert a JSON (IETF RFC-4627) string 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

9.88 kerf_type - Type Code

kerf_type(x)

Obtain a numeric typecode from a Kerf value.

```
KeRF> kerf_type 45.0
3
```

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
Zip	compressed [1,2]	zip	13

Kerf types

9.89 `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"
```

9.90 `last` - Last

`last(x)`

`last(count, x)`

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 count elements of `x`, repeating elements of `x` as necessary. Equivalent to **`take(-count, x)`**.

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

9.91 `left_join` - Left Join

`left_join(x, y, z)`

Perform a left join of the tables `x` and `y` on the column `z`. See **`Joins`**.

9.92 `len` - Length

`len(x)`

Determine the number of elements in `x`. Equivalent to **`count`**. Atomic elements have a count of 1. The length of a table is the number of rows it contains.

```
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  
KerF> len {{a:1 2 3 4 5}}  
5
```

9.93 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]
```

9.94 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]
```

9.95 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]
```

9.96 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. See **File I/O**.

9.97 ln - Natural Logarithm

`ln(x)`

Calculate $\log_e(x)$. Atomic.

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

9.98 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.

9.99 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]
```

9.100 `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]]
```

9.101 `map` - Make Map

`map(keys, values)`

Make a map from a list of keys and a list of values. These lists must be the same length. A map-based equivalent of `table`.

```
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}
```

9.102 `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
```

9.103 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]
```

9.104 max - Maximum

max(x)

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

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

9.105 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]
```

9.106 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]
```

9.107 median - Median

median(x)

Select the median of a list of numbers x:

```
KeRF> median 1 3 19 7 4 2
3.5
```

9.108 meta_table - Meta Table

meta_table(table)

Produce a table containing debugging metadata about some some table:

```
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

9.109 min - Minimum

min(x)

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

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

9.110 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]
```

9.111 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]
```

9.112 minv - Matrix Inverse

`minv(x)`

Calculate the inverse of a matrix x.

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

9.113 mkdir - Create directory

`mkdir(x)`

Create the directory specified by x.

```
KeRF> mkdir("dir/subdir")
```

9.114 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]
```

9.115 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, 0, 4]
```

9.116 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]]
```

9.117 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]
```

9.118 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]
```

9.119 **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 unary operator:

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

9.120 **negative** - Negative

`negative(x)`

Equivalent to **negate**.

9.121 **ngram** - N-Gram

`ngram(n, x)`

Break a list `x` into non-overlapping subsequences of length `n`.
Equivalent to `split(range(0, count(x), n), x)`.

```
KeRF> ngram(3, "Prisencolinensinainciusol")
["Pri", "sen", "col", "ine", "nsi", "nai", "nci", "uso", "l"]
KeRF> ngram(2, 34 12 44 19 29 90)
[[34, 12], [44, 19], [29, 90]]
```

9.122 **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]
```

9.123 `noteq` - Not Equal?

`noteq(x, y)`

A predicate which returns 1 if `x` is not equal to `y`. Equivalent to **`not equals`**. 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, .1000000000000001)
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]
```

9.124 `now` - Current DateTime

`now()`

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

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

9.125 `now_date` - Current Date

`now_date()`

Return a stamp containing the current date only in UTC.

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

9.126 `now_time` - Current Time

`now_time()`

Return a stamp containing the current time only in UTC.

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

9.127 `open_socket` - Open Socket

`open_socket(host, port)`

Establish a connection to a remote Kerf instance at hostname `host` and listening on `port` and return a connection handle. Both `host` and `port` must be strings. See **Network I/O**.

9.128 `open_table` - Open Table

`open_table(filename)`

Load a serialized table from the binary file `filename`. See **File I/O**.

9.129 `or` - Logical OR

`or(x, y)`

Calculate the logical *OR* of `x` and `y`. This operation is equivalent to `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]
```

9.130 `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]
```

9.131 `out` - Output

`out(x)`

Print a string `x` to standard output. See **General I/O**.

9.132 `parse_float` - Parse Float From String

`parse_float(string)`

Parse a string to obtain a floating point number. Atomic down to strings.

```
KeRF> parse_float "1"
1.0
KeRF> parse_float "2e4"
20000.0
KeRF> parse_float ["10","11","15"]
[10, 11, 15.0]
```

9.133 `parse_int` - Parse Integer From String

`parse_int(string)`

`parse_int(string, radix)`

Parse a string to obtain an integer. If no radix (numerical base) is provided, assume 10. The radix must be between 2 and 36, inclusive. Atomic over the left argument down to strings.

```
KeRF> parse_int "+337"
337
KeRF> parse_int("1010011", 2)
83
KeRF> parse_int("100", 16)
256
KeRF> parse_int(["Ab", "cD", "eF"], 16)
[171, 205, 239]
```

9.134 `parse_stamp` - Parse Timestamp From String

`parse_stamp(format, x)`

Parse a string to obtain a timestamp, according to the specified format. See **`format_stamp`** for details on permissible timestamp formats. Atomic over the right argument down to strings.

```
KeRF> parse_stamp("%S:%M:%H", "56:34:12")
12:34:56.000
KeRF> parse_stamp("%M", ["1", "2", "3"])
[00:01:00.000, 00:02:00.000, 00:03:00.000]
KeRF> f: "%H:%M:%S.%Q";
KeRF> t: now()
2016.05.23T00:26:31.688
KeRF> parse_stamp(f, format_stamp(f, t))
00:26:31.688
```

9.135 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]}
```

9.136 permutations - Permutations

`permutations(x)`

`permutations(x, repeats)`

Generate a list of all possible orderings of the elements of `x`. Normally this will operate on the unique elements of `x`, but if `repeats` is truthy all elements will be preserved:

```
KeRF> permutations(1 2 2)
      [[1, 2, 2],
       [2, 1, 2],
       [2, 2, 1]]
KeRF> permutations(1 2 2, 1)
      [[1, 2, 2],
       [1, 2, 2],
       [2, 1, 2],
       [2, 2, 1],
       [2, 1, 2],
       [2, 2, 1]]
```

If `x` is a map, operate on its keys:

```
KeRF> permutations({foo: 27, bar: 38})
      [{"foo", "bar"},
       {"bar", "foo"}]
```

9.137 plus - Plus

`plus(x, y)`

Equivalent to **add**.

9.138 pow - Exponentiation

pow(x)
pow(x, y)

Equivalent to `exp`.

9.139 powerset - Power Set

powerset(x)

Produce a list of all possible sublists of x. If x is a map, generate a powerset of its keys:

```
KeRF> powerset 45 67 33
[INT[],
 [33],
 [67],
 [67, 33],
 [45],
 [45, 33],
 [45, 67],
 [45, 67, 33]]
KeRF> powerset({foo: 23, bar: 94})
[[],
 ["bar"],
 ["foo"],
 ["foo", "bar"]]
```

9.140 rand - Random Numbers

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

Generate a random vector of x numbers from 0 up to but not including y.

```
KeRF> rand(10, 3)
[0, 2, 1, 2, 1, 2, 1, 2, 0, 2]
KeRF> rand(5, 3.0)
[0.74465, 1.72491, 0.79121, 2.53097, 0.573115]
```

If y is a list, select random elements from y.

```
KeRF> rand(6, "ABC")
"CBCBBA"
```

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

```
KeRF> rand(10)
1
KeRF> rand(10)
8
KeRF> rand(10.0)
6.77151
KeRF> rand(10.0)
0.401473
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]
```

9.141 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]
```

9.142 read_from_path - Read From Path

```
read_from_path(filename)
```

Load a serialized Kerf data structure from the binary file `filename`. See **File I/O**.

9.143 `read_parceled_from_path` - Read Parceled Table From Path

`read_parceled_from_path(path)`

Load a parceled Kerf data structure from the directory path. See **Parceled Tables**.

9.144 `read_striped_from_path` - Read Striped File From Path

`read_striped_from_path(path)`

Load a striped Kerf data structure from the directory path. See **Striped Files**.

9.145 `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)`.

9.146 `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 (see hashed)
G	IETF RFC-4122 UUID
N	IP address as parsed by C's <code>inet_pton()</code>
A	Nanoseconds since UNIX Epoch (Timestamp)
Z	Custom Datetime. (see <code>.Parse.strptime_format</code>)
Y	Custom Times. (see <code>.Parse.strptime_format2</code>)
*	Skipped field
W	64-bit (8 byte) compressed integer
4	32-bit (4 byte) compressed integer
2	16-bit (2 byte) compressed integer
1	8-bit (1 byte) compressed integer
V	64-bit compressed float
0	32-bit compressed float
D	64-bit compressed decimal (see notes)
9	compressed timestamp (see <code>.Parse.strptime_format</code>)
3	compressed STR32 (see notes)

Symbols accepted as part of fields

Columns with a **compressed** datatype are compressed into memory as the file is parsed. The STR32 type is capable of representing a vector of strings no longer than 32 characters, which may not contain the null character \0. The decimal type D is stored as a fixed-point integer, retaining 4 decimal places. 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

As another example, let's take a look at how some specialized types are handled. Observe how strings in the STR32 column are truncated as necessary, and the decimal column retains only the first 4 decimal places:

```
Label&Float
First&10.5
Second&0.123456
Third&90000
A considerably longer string which won't fit.&320
```

```
KeRF> t: read_table_from_delimited_file("&", "delimited.txt", "3D", 1)
```

Label	Float
First	10.5
Second	0.1234
Third	90000.0
A considerably longer string whi	320.0

```
KeRF> t["Label"]
COMPRESSED["First", "Second", "Third", "A considerably longer string whi"]
KeRF> kerf_type_name t["Label"]
"zip"
KeRF> kerf_type_name t["Float"][2]
"float"
```

9.147 `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
<code>fields</code>	String	No	As in <code>read_table_from_delimited_file</code> .
<code>widths</code>	Integer Vector	No	The width of each column in characters.
<code>line_limit</code>	Integer	Yes	The maximum number of rows to load.
<code>titles</code>	List of String	Yes	Key for each column in the resulting table.
<code>header_rows</code>	Integer	Yes	How many rows are treated as column headers.
<code>newline_separated</code>	Boolean	Yes	if false, do not expect newlines separating rows.

Settings described in attributes

Symbol	Datatype
R	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

9.148 `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)`.

9.149 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\""
```

9.150 repeat - Repeat

`repeat(n, x)`

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

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

9.151 reserved - Reserved Names

`reserved()`

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

9.152 reset - Reset

`reset()`

`reset(drop_args)`

Reset the Kerf interpreter, clearing the workspace and cleaning up any open resources. If a truthy argument is supplied, reset the interpreter as if invoked without any command-line arguments.

```
> ./kerf -q
Kerf> a: 437;
Kerf> reset()
Kerf> a

Undefined token error

Kerf>
```

9.153 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"
```

9.154 rsum - Running Sum

`rsum(x)`

Calculate a running sum of the elements of the list `x`, from left to right. nans 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 []
[]
```

9.155 run - Run

`run(filename)`

Load and run Kerf source from a file. Equivalent to **load**.

9.156 **search** - Search

`search(x, y)`

Look for `x` in `y`. If found, return the index. If not found, return `NAN`:

```
KeRF> search(3, 0 3 17 30)
1
KeRF> search(30, 0 3 17 30)
3
KeRF> search(15, 0 3 17 30)
NAN
KeRF> search("F", "AEIOU")
NAN
```

9.157 **seed_prng** - Set random seed

`seed_prng(x)`

Sets random number generator seed used in **rand** to `x`.

9.158 **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. See **Network I/O**.

9.159 **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. See **Network I/O**.

9.160 **setminus** - Set Disjunction

`setminus(x, y)`

Equivalent to **except**.

9.161 **shell** - Shell Command

`shell(x)`

Execute a string `x` containing a shell command as if from `/bin/sh -c x`. See **General I/O**.

9.162 **shift** - Shift

```
shift(n, x)
shift(n, x, fill)
```

Offset the list `x` by `n` positions, filling shifted-in positions with `fill`.

```
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 `fill` 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]
```

9.163 **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"
```

9.164 **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]
```

9.165 **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]
```

9.166 **sleep** - Sleep

`sleep(x)`

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

```
KeRF> timing 1;
KeRF> sleep 50
50
51 ms
KeRF>
```

9.167 **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"
```

Sort when run on a table sorts in column order, so for maximum performance on time oriented joins, put the time column first in the table.

9.168 `sort_debug` - Sort Debug

`sort_debug(x)`

Return a map containing debugging information about the list or table `x` which is relevant to the performance and internal datapaths of searching and sorting. If the input is a table which is actually sorted, it will set global `attr.sorted` flag if not already set.

```
KeRF> sort_debug range(4)
{attr_sorted:1, is_array:1, is_enum:0, is_actually_sorted_array:1, is_index:0,
 is_index_working:0, is_table:0, is_table_sorted:0}

KeRF> sort_debug "ACED"
{attr_sorted:0, is_array:1, is_enum:0, is_actually_sorted_array:0, is_index:0,
 is_index_working:0, is_table:0, is_table_sorted:0}

KeRF> sort_debug 27
{attr_sorted:1, is_array:0, is_enum:0, is_actually_sorted_array:0, is_index:0,
 is_index_working:0, is_table:0, is_table_sorted:0}

KeRF> sort_debug {{a: 4 2 1}}
{attr_sorted:0, is_array:0, is_enum:0, is_actually_sorted_array:0, is_index:0,
 is_index_working:0, is_table:1, is_table_sorted:0}
```

For tables, the sort attribute is for the first column of the table.

9.169 `split` - Split List

`split(x, y)`

Subdivide the list `y` at the index/indices given by `x`. Equivalent to `y[xvals part rsum (xkeys y) in x]`.

```
KeRF> split(4 9, "SomeWordsTogether")
["Some", "Words", "Together"]
KeRF> 3 split "ABCDEF"
["ABC", "DEF"]
KeRF> split(0 1 3 5, 11 22 33 44 55)
[[11], [22, 33], [44, 55]]
```

9.170 `sqrt` - Square Root

`sqrt(x)`

Calculate the square root of `x`. Atomic.

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

9.171 stamp - Cast to Stamp

stamp(x)

Cast x to a timestamp. Atomic down to strings. To convert a string to a timestamp using a custom format, see **parse_stamp**.

```
KeRF> stamp []
STAMP []
KeRF> stamp [0, "2001.02.03"]
[00:00:00.000, 2001.02.03]
```

9.172 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
```

9.173 std - Standard Deviation

std(x)

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

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

9.174 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"]}
```

9.175 subtract - Subtract

subtract(x, y)

Equivalent to **minus**.

9.176 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
```

9.177 table - Make Table

`table(columns, values)`

Make a table from a list of column names and a matrix of values. A table-based equivalent of `map`.

```
KeRF> table(["foo", "bar"], [1 2 3, 2001.1.1 2001.1.2 2001.1.3])
```

foo	bar
1	2001.01.01
2	2001.01.02
3	2001.01.03

9.178 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"]
```

9.179 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**.

```
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"
```

9.180 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]
```

9.181 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]
```

9.182 **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]
```

9.183 **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
```

9.184 **tolower** - To Lowercase

`tolower(x)`

Convert a string x to lowercase. Equivalent to **floor**.

9.185 **toupper** - To Uppercase

`toupper(x)`

Convert a string x to uppercase. Equivalent to **ceil**.

9.186 **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]]
```

9.187 **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"]
```

9.188 **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
```


9.194 **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 unary operator:

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


9.195 `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)`.

9.196 `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> shell("wc -c example.psv")
["      23 example.psv"]
KeRF> shell("cat example.psv")
["a|b", "1|one", "2|two", "3|three"]
```

9.197 `write_stripped_to_path` - Write Striped File To Path

```
write_stripped_to_path(path, x)
```

Write x to disk at path as a striped file. See **Striped Files**.

9.198 `write_text` - Write Text

```
write_text(filename, x)
```

Write the value x to a text file filename, creating the file as necessary. See **File I/O**.

9.199 `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. See **File I/O**.

9.200 xkeys - Object Keys

$$\text{xkeys}(x)$$

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"]
```

9.201 xvals - Object Values

$$\text{xvals}(x)$$

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]]
```

9.202 zip - Compress Object

$$\text{zip}(x)$$

Produce a string representing a compressed version of `x` suitable for reconstruction with `unzip`.

```
KerF> count 2048 take "ABCD"
      2048
KerF> count zip 2048 take "ABCD"
      1316
KerF> zip 2048 take "ABCD"
      "{{" "sJ\u0000\u0000\u0000\u00ca\u0000\u0000\u0000\u00ca\u0000\u0000\u0000\u0000..."
```

10 Combinator Reference

10.1 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**.

10.2 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**.

10.3 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**.

10.4 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]
```

10.5 mapcores - Map to Cores

Similar to `mapdown`, except it distributes work to multiple CPU cores (as available) and performs work in parallel. There is some overhead to distributing work and collecting results, so only use `mapcores` after careful profiling. For a realistic use case, see the discussion of optimizing a **Base64 Encoder**.

```
KeRF> time: {[f] s:now(); v:f(); [v, stamp_diff(now(), s)]}];

KeRF> time {[ ] {[x] sleep 100; x*2} mapdown 1 3 7 9}
[[2, 6, 14, 18], 404506000]

KeRF> time {[ ] {[x] sleep 100; x*2} mapcores 1 3 7 9}
[[2, 6, 14, 18], 184118000]
```

Every parallel worker is given its own independent environment tree, and writes to global variables are discarded after results are gathered. Side effects to IO devices are performed in an arbitrary order unless otherwise synchronized.

```
KeRF> a: [0];
KeRF> {[x] a[0]:2*x; a[0]} mapdown 1 3 7 9
[2, 6, 14, 18]
KeRF> a
[18]

KeRF> a: [0];
KeRF> {[x] a[0]:2*x; a[0]} mapcores 1 3 7 9
[2, 6, 14, 18]
KeRF> a
[0]

KeRF> {[x] display x; 2*x} mapcores 1 3 7 9
3
7
9
1
[2, 6, 14, 18]
```

Uses of `mapcores` can be nested, generally with diminishing returns. The current implementation arbitrarily caps nesting at 2:

```
KeRF> {[x] {[x] 2*x} mapcores range x} mapcores 3 2 5
[[0, 2, 4],
 [0, 2],
 [0, 2, 4, 6, 8]]

KeRF> {[x] {[x] {[x] 2*x} mapcores range x} mapcores range x} mapcores 2 3
...nge x} mapcores range x} mapcores 2 3
~
Parallel execution error
```

10.6 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]
```

10.7 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**.

10.8 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**.

10.9 reconverge - Reconverge

Equivalent to **unfold**.

10.10 reduce - Reduce

Equivalent to **fold**.

10.11 refold - Refold

Equivalent to **unfold**.

10.12 rereduce - Re-Reduce

Equivalent to **unfold**.

10.13 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**.

11 Global Reference

11.1 Environment

11.1.1 `.Argv` - Arguments

A list of the arguments provided to Kerf at the command line.

```
>kerf -q foo
File handle status failed during directory check.: No such file or directory
KeRF> .Argv
["kerf", "-q", "foo"]
```

11.1.2 `.Help` - Function Reference

A table which can be examined at the command line, listing subject, name of item of interest, usage and description. See `help` for more information.

11.2 Math

11.2.1 `.Math.BILLION` - Billion

Constant representing $[10^9]$.

11.2.2 `.Math.E` - **E**

Constant representing Euler's number. 2.7182818284590452353602.

11.2.3 `.Math.TAU` - **Tau**

Constant representing 2π . 6.2831853071795864769252.

11.3 Net

11.3.1 `.Net.client` - **Client**

During IPC execution, contains a constant representing the current client's unique handle. See **Network I/O**.

11.3.2 `.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. See **Network I/O**.

11.3.3 `.Net.parse_request` - **Parse Request**

An HTTP server will call this single-argument function with the text of a completed GET request. The return value will be passed to the browser, provided the type is character vector. See **Network I/O**.

11.4 Parse

11.4.1 `.Parse.strptime_format` - Time Stamp Format

Specifies the format used for formatting and parsing dates and time stamps from delimited files when the field specifier is Z or 9. Builds directly on the standard C function `strptime`. See **`format_stamp`** for details.

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

11.4.2 `.Parse.strptime_format2` - Time Format

Specifies the format used for formatting and parsing timestamps from delimited files when the field specifier is Y. Used for parsing time-only columns in data files where dates and times are in separate columns.

11.5 Print

11.5.1 `.Print.stamp_format` - Print Stamp Format

If set, specifies the format used for printing timestamps. See **`format_stamp`** for details. Setting this global is particularly useful if you want to see a finer-grained display of timestamps which includes nanoseconds.

12 Programming Techniques

This section contains case studies illustrating how Kerf can be used to solve problems, ranging from simple to complex. We will build up solutions step by step and consider tradeoffs in performance and style.

12.1 Reversing a Map

A map links a set of keys with a set of values. It is easy and efficient to use a key to look up the associated value. Sometimes it is desirable to go the other way- using a value to look up the associated key. One way to approach this problem might be to iterate through each key in the map until we found one which is associated with our target value:

```
def find_key(m, val) {
  keys: xkeys m
  for(i: 0; i < len(keys); i: i+1) {
    if (m[keys[i]] == val) { return keys[i] }
  }
  return null
}
```

As the number of keys in the map scales up, the amount of time this function takes to run increases proportionally. If we need to perform many repeated reverse lookups against a large map, this approach will be prohibitively slow.

As a general rule of thumb when programming, if something is too expensive to calculate, cache it. By writing a function which analyzes a map and constructs a new map which “reverses” the keys and values of the original, we can pay this construction cost once and then achieve efficient reverse-lookups.

There are several ways to approach this problem. A programmer used to imperative programming languages might come up with something like this:

```
def map_reverse_1(m) {
  r: {}
  keys: xkeys m
  vals: xvals m
  for(i: 0; i < len(keys); i: i+1) {
    r[vals[i]]: keys[i]
  }
  return r
}
```

Perhaps you want to get fancy and try to represent the same algorithm using the combinator **fold**?

```
{ } { [m, e] m[e[0]]: e[1] } fold transpose([xvals m, xkeys m])
```

The simplest approach is to take advantage of the **map** built-in function:

```
def map_reverse_2(m) {
  return map(xvals m, xkeys m)
}
```

All of these solutions are making an unsafe assumption. The keys of a map are always unique by definition, but what happens if the values are not unique?

```
KeRF> map_reverse_2({a: "A", b:"B", c:"C"})
{A:"a", B:"b", C:"c"}
KeRF> map_reverse_2({a: "A", b:"B", c:"A"})
{A:"a", B:"b"}
```

We've lost information! A more robust map reversal routine should gather the keys of repeated values as a list, producing a result more like:

```
KeRF> map_reverse_3({a: "A", b:"B", c:"C"})
{A:["a","c"], B:["b"]}
```

First, let's look at an imperative approach to solving the problem. We can iterate through the keys of the original map, as before. Instead of storing the key directly in the result map, we will append it to a list stored in the map. This requires us to first ensure that any slot in the result map is initialized with an empty list:

```
def map_reverse_3(m) {
  r: {}
  keys: xkeys m
  vals: xvals m
  for(i: 0; i < len(keys); i: i+1) {
    if (not r has_key vals[i]) { r[vals[i]]: [] }
    r[vals[i]]: join(r[vals[i]], enlist keys[i])
  }
  return r
}
```

The **enlist** is important here- without it, string keys would be mashed together:

```
KeRF> join([], "foo")
"foo"
KeRF> join("foo", "bar")
"foobar"
KeRF> join([], enlist "foo")
["foo"]
KeRF> join(["foo"], enlist "bar")
["foo", "bar"]
```

We can simplify this slightly by using “spread assignment” to initialize our result map with empty lists in every entry:

```
...
r[vals]: [[]]
for(i: 0; i < len(keys); i: i+1) {
  r[vals[i]]: join(r[vals[i]], enlist keys[i])
}
...
```

What happens if we leave out this initialization entirely? Try it!

Now let's look at a functional approach to solving this problem. If we partition the value set of the original map, we obtain lists of the indices of the values which are identical:

```
KerF> m: {a: "A", b:"B", c:"A"}  
      {a:"A", b:"B", c:"A"}  
KerF> xvals m  
      ["A", "B", "A"]  
KerF> part xvals m  
      {A:[0, 2], B:[1]}
```

Since the key and value vectors produced by **xkeys** and **xvals** line up, we could also use those indices to obtain the corresponding keys for each group of values. Kerf's powerful indexing facilities make this simple:

```
KerF> xkeys m  
      ["a", "b", "c"]  
KerF> xvals part xvals m  
      [[0, 2], [1]]  
KerF> (xkeys m)[xvals part xvals m]  
      [{"a", "c"}, [{"b"}]]
```

Now we have what we need to assemble the result, so we use **map** to join the unique elements of the original value set with the lists of corresponding keys we computed previously. The **unique** function collects results in the order they appear, just like **part**, so everything will line up properly:

```
def map_reverse_4(m) {  
    nk: unique xvals m  
    nv: (xkeys m)[xvals part xvals m]  
    return map(nk, nv)  
}
```

Notice how we were able to use the Kerf REPL and a single working example to experiment interactively and then distill the results into a general definition. It is very natural to develop functional solutions to problems in this manner. Manipulating an entire data structure at once often results in a simpler solution with fewer conditionals and loops than trying to solve a problem “one step at a time”.

It is also possible to write this as a more compact one-liner by avoiding intermediate variables and using some of the symbolic shorthands for **map** (!), **unique** (%), **enumerate** (^) and **part** (&), but the result is much harder to read and understand at a glance:

```
{[m] (%xvals m)!(^m)[xvals(&xvals m)]}
```

Shorthand is a nice way to save typing at the REPL, but favor a more relaxed, verbose style when writing larger programs- future maintainers (yourself included) will be thankful! Use Kerf's syntactic alternatives- infix function calls, optional parentheses, function aliases, etc.- to write your programs in the clearest, most readable way possible.

12.2 Run-Length Encoding

Run-Length Encoding (RLE) is a simple means of compressing data. Wherever there is a repeated run of identical elements, instead of storing each element store a single copy of the element and the length of the run. We will use this problem as an opportunity to gain familiarity with more of Kerf's combinators and the elegance of tacit definitions.

If our input list looked like:

```
"AAABBAAAAAAAA"
```

An RLE-compressed representation might look like:

```
[[3, `A"], [2, `B"], [7, `A"]]
```

Let's start by writing a *decompressor*- given an RLE-compressed list, reconstruct the original. For a given element of the list, we can use **take** to create a run:

```
KeRF> p: [3, `A"]
      [3, `A"]
KeRF> take(p[0], p[1])
      "AAA"
```

One way to apply a binary function to a pair of arguments is to use the combinator **fold**. A fold is like placing a function between the elements of a list. Thus, these two expressions are equivalent:

```
KeRF> take fold p
      "AAA"
KeRF> p[0] take p[1]
      "AAA"
```

We really want to apply **take** to *each* pair in the original list. The combinator **mapdown** applies a unary function to each element of a list, returning a list of results:

```
KeRF> (take fold) mapdown [[3, `A],[2, `B],[7, `A]]
      ["AAA", "BB", "AAAAAAA"]
```

The parentheses are not required- in this case they are simply making it clearer that the function **mapdown** is applying to each element of the list to the right is "**take fold**". When combinators are attached to functions, they form new compound functions which can be passed around or stored in variables just like any other function:

```
KeRF> take fold mapdown [[3, `A],[2, `B],[7, `A]]
      ["AAA", "BB", "AAAAAAA"]
KeRF> take fold
      take fold
```

To reproduce the original sequence from this list of runs, we need to join all the runs together. The built-in function **flatten** does exactly what we need. Alternatively, we could use the equivalent **join fold**:

```
KeRF> flatten ["AAA", "BB", "AAAAAAA"]
      "AAABBAAAAAAAA"
KeRF> join fold ["AAA", "BB", "AAAAAAA"]
      "AAABBAAAAAAAA"
```

Putting everything together, we could write a definition for `rle_decode` in several ways:

```
def rle_decode(x) {return flatten take fold mapdown x}

rle_decode: {[x] flatten take fold mapdown x}

rle_decode: flatten take fold mapdown
```

This last approach is called a *tacit definition*- it doesn't require naming any variables or arguments. The combinators glue together functions to produce a function that is simply waiting for a right argument. Composing together functions won't always work out this nicely, but when a tacit definition is possible the results can be very aesthetically pleasing, as if there were hardly any syntax to the language at all:

```
KeRF> rle_decode: flatten take fold mapdown
      flatten take fold mapdown

KeRF> rle_decode [[3, `A],[2, `B],[7, `A]]
      "AAABBAAAAAAAA"

KeRF> flatten take fold mapdown [[3, `A],[2, `B],[7, `A]]
      "AAABBAAAAAAAA"
```

Now that we're a bit more comfortable with using combinators, let's tackle the *compressor*- given a list, we want to break it into runs. For each run, we need to determine the first element (or any element, really) and the length of the run.

To find the start of each run, we can compare each element of the list with the item which came before it. If they're different, we are beginning a new run. Otherwise, we're continuing an existing run. The combinator **mapback** applies a function to each element of a list and its predecessor, which is just what we're looking for:

```
KeRF> != mapback "AAABBAAAAAAAA"
      [ ` "A", 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0 ]
```

Well, *nearly* what we're looking for. What's that character doing at the beginning of our result? **mapback** doesn't have anything to pair the first element of a list up with, so by default it leaves that item alone. If we supply a value on the left, **mapback** will use that to pair with the first element of the list on the right. Supplying a null will ensure that the first list element is considered "not equal to" this default value:

```
KeRF> `A != mapback "AAABBAAAAAAAA"
      [ 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0 ]

KeRF> `B != mapback "AAABBAAAAAAAA"
      [ 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0 ]

KeRF> `B != mapback "BAABBAAAAAAAA"
      [ 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0 ]

KeRF> null != mapback "AAABBAAAAAAAA"
      [ 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0 ]
```

The function **which**, given a boolean list, produces a list of the indices of the 1s. If we index into the original list, we can retrieve a list of the items which begin each run, in order:

```
KeRF> s: "AAABBAAAAAAAA"
      "AAABBAAAAAAAA"
KeRF> null != mapback s
      [1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
KeRF> which null != mapback s
      [0, 3, 5]
KeRF> s[which null != mapback s]
      "ABA"
```

Now we just need to find the lengths of each run. Let's consider that vector we already created which identifies run heads. If we take a running sum (**rsum**) of that list, we can uniquely label all the members of each run:

```
KeRF> h: null != mapback s
      [1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
KeRF> rsum h
      [1, 1, 1, 2, 2, 3, 3, 3, 3, 3, 3, 3]
```

Partitioning these labels will conveniently group them together, and we can count each of the resulting groups:

```
KeRF> part rsum h
      1:[0, 1, 2], 2:[3, 4], 3:[5, 6, 7, 8, 9, 10, 11]
KeRF> count mapdown part rsum h
      [3, 2, 7]
```

An alternative to **part rsum** would be to use **split**:

```
KeRF> split(which h, s)
      ["AAA", "BB", "AAAAAAA"]
KeRF> count mapdown split(which h, s)
      [3, 2, 7]
```

With the list of counts and the list of values, we can join each to produce the tuples we want. By giving the combinator **mapdown** a left argument and a binary function, we can neatly “zip” together our lists:

```
KeRF> 1 2 3 join mapdown "ABC"
      [[1, `A"],
       [2, `B"],
       [3, `C"]]
```

Bringing everything together,

```
def rle_encode(s) {
  heads:  null != mapback s
  vals:   s[which heads]
  counts: count mapdown part rsum heads
  return  counts join mapdown vals
}
```

For the sake of comparison, here's an imperative solution:

```
def rle_encode_2(s) {
  r: []
  c: 1
  for(i: 0; i < len(s); i: i+1) {
    if (s[i] != s[i+1]) {
      r: join(r, enlist [c, s[i]])
      c: 1
    } else {
      c: c+1
    }
  }
  return r
}
```

Which of these is easier to understand and reason about? The imperative program is familiar, but involves a number of variables which continuously change throughout execution. Did we make any off-by-one errors? The functional solution replaces loops and conditionals with built-in functions and combinators, and never changes the contents of a variable once it is assigned.

“Readability” is inherently subjective, so let's compare performance:

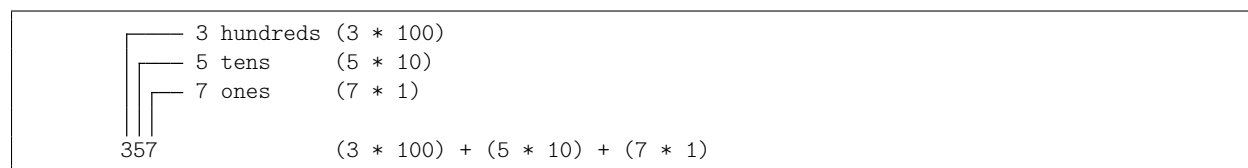
```
KerF> data: 10000?"ABCD";
KerF> timing 1;
KerF> rle_encode data ;
      21 ms
KerF> rle_encode_2 data ;
      781 ms
```

Kerf executes the functional solution substantially faster. Combinators and built-in functions which operate on entire lists at once are highly optimized and have very little interpreter overhead, so an individual operation can approach native execution speeds. Explicit loops and conditionals force the interpreter to do most of the heavy lifting, which is less efficient.

12.3 Base64 Conversion

Base64 is an encoding scheme which permits storing arbitrary binary data- often ASCII text- using a restricted set of 64 symbols. Base64 encoded data avoids invisible control characters and common delimiters, making it easy to embed inside other document formats or copy and paste between applications without getting scrambled or truncated along the way. In this section we will develop a Base64 encoder and decoder in Kerf, and along the way learn how to dramatically improve the performance of parallel operations by using the `mapcores` combinator.

First, let's review numeric base conversion in general. The numbers we see everywhere on a daily basis use what's called a *positional number system*. In this system, the value of each digit is determined by its position in the overall number. Digits further to the left are "worth more" than digits on the right:



With 10 distinct digits (0-9), each successive place to the left is worth 10 times as much as the previous. We call a number system which has 10 distinct digits *base-10* or *decimal*. A number system which only has two distinct digits could be called *base-2* or *binary*.

Representing a number in different bases can be useful for a variety of applications. Let's begin by developing a routine which can decompose a Kerf integer into its digits:

```
Kerf> number_digits(357)
[3, 5, 7]
```

The most obvious approach is to reach for string manipulation. If we cast a number to a string with `rep`, the digits will be successive characters:

```
Kerf> rep 357
"357"
```

To obtain the ordinal value of digits, we could convert them to ASCII character codes and subtract the ASCII value of "0" (which happens to be 48) or we could simply `eval` each character:

```
Kerf> (int "357")-48
[3, 5, 7]
Kerf> eval enlist mapdown "357"
[3, 5, 7]
```

This works fine for base-10, since that is the base Kerf naturally uses for displaying numbers. A string-oriented approach won't help us if we want to support an *arbitrary* base. A more general solution considers the value of each digit's place, as depicted above.

Let's try to calculate the value of each place in a 4-digit base-10 number. There are a number of reasonable approaches- can you think of any aside from those shown here?

```
Kerf> v: reverse floor 10 pow range 4           // pow's result is always a float, so we use floor
[1000, 100, 10, 1]
Kerf> v: reverse (4-1) {[x] 10*x} unfold 1      // using the 'repeat N times' form of unfold
[1000, 100, 10, 1]
```

If we have each place value in v , dividing the original number by v and then taking the result modulo the base will determine the digit in each place:

```
KeRF> floor 357 / v
[0, 3, 35, 357]
KeRF> (floor 357 / v) % 10
[0, 3, 5, 7]
```

Generalizing,

```
def unpack_digits(base, places, n) {
  v: reverse floor base pow range places
  return (floor n / v) % base
}
```

```
KeRF> unpack_digits(10, 3, 357)
[3, 5, 7]
KeRF> unpack_digits(2, 8, 61)
[0, 0, 1, 1, 1, 1, 0, 1]
```

As written, this routine needs to be given the length of the number in digits ahead of time. The **log** with respect to the base (rounding up) can determine how many digits we need to represent the input number in that base:

```
def digits(base, n) {
  v: reverse floor base pow range ceil log(base, n)
  return (floor n / v) % base
}
```

```
KeRF> digits(10, 65539)
[6, 5, 5, 3, 9]
KeRF> digits(10, 15)
[1, 5]
KeRF> digits(16, 255)
[15, 15]
```

If we have digits and a base, getting back to a number is extremely easy- multiply the digits by their place value and take the sum of the results:

```
KeRF> 3 5 7 * 100 10 1
[300, 50, 7]
KeRF> sum 3 5 7 * 100 10 1
357
```

Using a similar pattern as before, we'll create a generalized routine for doing this in any base:

```
def pack_digits(base, n) {
  v: reverse floor base pow range count n
  return sum n * v
}
```

Base64 encodes 3 bytes of 8-bit ASCII into 4 6-bit characters like so:

ASCII char	(D								E							
ASCII value	40								68								69							
Binary	0	0	1	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1
Index	10								4								17							
Base64	K								E								R							

The characters used to represent the 64 distinct values of each 6-bit chunk vary from implementation to implementation, but a common scheme uses the uppercase alphabet followed by the lowercase alphabet, the digits 0 through 9 and finally the characters + and /:

```
KeRF> int "AZaz09+/"
[65, 90, 97, 122, 48, 57, 43, 47]
KeRF> b64: char range(65, 91)#range(97, 123)#range(48, 58)#43#47
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+\/"
KeRF> count b64
64
```

Using the definitions we've written so far, it is straightforward to do a conversion from ASCII into Base64 following the steps shown in the figure above. Convert characters to indices with **int**, break those indices into binary digits, **flatten** the list of bits and re-slice it into 6-bit fields with **ngram**, convert back to decimal and then index through the lookup table we just built. Step by step,

```
KeRF> int "(DE"
[40, 68, 69]
KeRF> {[x] unpack_digits(2, 8, x)} mapdown int "(DE"
[[0, 0, 1, 0, 1, 0, 0, 0],
 [0, 1, 0, 0, 0, 1, 0, 0],
 [0, 1, 0, 0, 0, 1, 0, 1]]
KeRF> b: flatten {[x] unpack_digits(2, 8, x)} mapdown int "(DE"
[0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1]
KeRF> 6 ngram b
[[0, 0, 1, 0, 1, 0],
 [0, 0, 0, 1, 0, 0],
 [0, 1, 0, 0, 0, 1],
 [0, 0, 0, 1, 0, 1]]
KeRF> {[x] pack_digits(2, x)} mapdown 6 ngram b
[10, 4, 17, 5]
KeRF> b64[ {[x] pack_digits(2, x)} mapdown 6 ngram b]
"KERF"
```

Generalizing,

```
def to_base64_simple(s) {
  b: flatten {[x] unpack_digits(2, 8, x)} mapdown int s
  return b64[ {[x] pack_digits(2, x)} mapdown 6 ngram b]
}
```

One important detail we haven't handled yet is what happens when we have an input string which is not divisible by 3 bytes. Base64 pads such data with zeroes out to a full 6-bit character boundary:

```
KeRF> (6 - count 1 0) % 6
4
KeRF> (6 - count 1 0 1 1) % 6
2
KeRF> (6 - count 1 0 1 1 0 1) % 6
0
```

If we had to add 2 bits, we'll add = to the end of the output string. If we had to add 4 bits, we'll add == to the end of the output string.

```
def to_base64(s) {
  bin: flatten {[x] unpack_digits(2, 8, x)} mapdown int s
  pad: (6 - count bin) % 6
  bip: bin join take(pad, 0)
  enc: b64[ {[x] pack_digits(2, x)} mapdown 6 ngram bip]
  return enc join take(pad / 2, `="`)
}
```

```
KeRF> to_base64 "A stri"
"QSBzdHJp"
KeRF> to_base64 "A strin"
"QSBzdHJpbg=="
KeRF> to_base64 "A string"
"QSBzdHJpbmc="
KeRF> to_base64 "A string."
"QSBzdHJpbmcu"
```

If Base64 encoding is a performance-critical component of your application, you might want to consider calling out to an existing C library to do the conversion. We'll discuss Kerf's FFI and how to do this kind of enhancement in a later section. In the shorter term, though, there are some clear opportunities for speeding up our current solution. The following complete implementation specializes and inlines the logic we created in `unpack_digits` and `pack_digits` so that we aren't constantly reconstructing digit place tables:

```
b64: char range(65, 91)#range(97, 123)#range(48, 58)#43#47
bits_8: reverse floor 2 pow range 8
bits_6: 2 drop bits_8

def to_base64_fast(s) {
  bin: flatten {[x] (floor x / bits_8) % 2} mapdown int s
  pad: (6 - count bin) % 6
  bip: bin join take(pad, 0)
  enc: b64[{:x sum x * bits_6} mapdown 6 ngram bip]
  return enc join take(pad / 2, `=")
}
```

How much more efficient is this approach?

```
KerF> data: 50000 ? "ABCDE";
KerF> timing 1;
KerF> to_base64 data;
      3.2 s
KerF> to_base64_fast data;
      2.5 s
KerF> 1 - 2.5/3.2
      0.21875
```

Just over 20% faster! Remember- the best way to make code fast is to make it *do less*. If you can cache something (as we are with `bits_8` and `bits_6`), you can avoid recomputing it frequently.

Another option would be to observe that both instances of **mapdown** are performing large, entirely parallel operations. Kerf has an extremely handy facility for improving performance in these situations called **mapcores**. This combinator will automatically distribute work across multiple CPU cores and then gather the results. All we have to do is replace **mapdown** with **mapcores**:

```
def to_base64_cores(s) {
  bin: flatten {[x] (floor x / bits_8) % 2} mapcores int s
  pad: (6 - count bin) % 6
  bip: bin join take(pad, 0)
  enc: b64[{:x sum x * bits_6} mapcores 6 ngram bip]
  return enc join take(pad / 2, `=")
}
```

On a humble 2 core laptop, this tweak allows our program to run in *five percent the original runtime!* Your mileage in other applications may vary. **mapcores** has some overhead, so if you're working with very small datasets it could slow down execution.

```
KerF> to_base64_cores data;
      188 ms
KerF> to_base64_cores "ABCDABCDABCD";
       3 ms
KerF> to_base64_fast "ABCDABCDABCD";
       2 ms
```

For the sake of completeness, let's briefly discuss transforming base64 encoded data back into its original form. If we ignore padding, we can use a very similar approach to our encoder. Before we could simply obtain the ASCII values of characters by using **int**, but this time around we need to use **search** to look up each character in the b64 table. The rest is familiar: convert to binary, **flatten** and re-slice with **ngram**, convert back to ASCII indices.

```
def from_base64_simple(s) {
  ind: s search mapleft b64
  bin: flatten {[x] (floor x / bits_6) % 2} mapcores i
  enc: {[x] sum x * bits_8} mapcores 8 ngram b
  return char enc
}
```

If the input string contains the padding character =, **search** will return NAN:

```
KerF> "Foo=" search mapleft b64
[5, 40, 40, NAN]
```

We can count how many padding characters (if any) were found with **count_null**. Then we can convert the NANs to 0 for the rest of processing by cleverly employing **msum**.

```
KerF> count_null [5, 40, 40, NAN]
1
KerF> 1 msum [3, 7, NAN, 15, NAN]
[3, 7, 0, 15, 0]
```

When we're done handling padding and performing the normal conversion process, all that remains is to **drop** the padding characters from the end of the result:

```
def from_base64(s) {
  ind: s search mapleft b64
  pad: count_null ind
  inp: floor 1 msum ind
  bin: flatten {[x] (floor x / bits_6) % 2} mapcores inp
  enc: {[x] sum x * bits_8} mapcores 8 ngram bin
  return drop(-pad, char enc)
}
```

```
KerF> from_base64("QSBzdHJpbg==")
"A strin"
KerF> from_base64("QSBzdHJpbmc=")
"A string"
KerF> from_base64("QSBzdHJpbmcu")
"A string."
```

We've written a program which performs a reasonably complex format conversion without any explicit loops or conditionals. The logic is uniform, easy to follow and easy to test. Kerf's **mapcores** provided us with a painless drop-in replacement for **mapdown** which takes advantage of the inherent parallelism in our code.

12.4 HTTP Fetching

KeRF is designed for processing and analyzing data. Many websites exist today which expose access to interesting datasets via an HTTP (HyperText Transfer Protocol) interface. Let's look at HTTP and the performance implications of different styles of data import. We will also take this as an opportunity to explore calling out to C functions using Kerf's Foreign Function Interface.

We will use <https://www.quandl.com> for our examples. Quandl provides a wide variety of financial and economic datasets, many of which are available free of charge. Creating an account with Quandl and using credentials with API requests increases the number of queries you are allowed to perform daily, but the system is usable without any form of registration.

In an HTTP transaction, a client makes a *request* for a resource, and the server provides a *response*. Requests have an associated *verb*, which indicates what the client wishes the server to do. The most basic type of request verb is GET- a request for the retrieval of a named resource, indicated by a URL. The response will contain a *response code* indicating success or a class of failure which occurred, and sometimes a *payload*. HTTP is stateless: each request-response pair is an independent transaction, and requests must include all context necessary for processing.

The simplest way to perform an HTTP request is via the Unix curl utility. By using the **shell** function, we can invoke curl from Kerf. In this example, using the .json suffix on the URL instructs Quandl to produce a response in the JSON format, which we can easily parse. **shell** breaks input on newlines, so it is necessary to join these back together via **implode**. The -s option prevents curl from printing information about its download progress, which is not relevant for us.

```
KeRF> url: "https://www.quandl.com/api/v3/datasets/WIKI/AAPL/data.json?rows=10";
KeRF> d: kerf_from_json implode(`\n`, shell "curl -s "#url#" -X GET");
```

It may also be a good idea to use the -o flag option with curl, which makes it write the response to a temporary file for later retrieval:

```
>curl -s -o tempfile.dat https://url.com/path -X GET
```

The response contains a great deal of potentially useful metadata. We can obtain column names:

```
KeRF> d["dataset_data"]["column_names"]
["Date", "Open", "High", "Low", "Close", "Volume", "Ex-Dividend", "Split Ratio", "Adj. Open",
"Adj. High", "Adj. Low", "Adj. Close", "Adj. Volume"]
```

The data itself is represented in a row-oriented fashion:

```
KeRF> 2 first d["dataset_data"]["data"]
[["2015-12-08", 117.52, 118.6, 116.86, 118.23, 34086875.0, 0.0, 1.0, 117.52, 118.6, 116.86,
118.23, 34086875.0],
["2015-12-07", 118.98, 119.86, 117.81, 118.28, 31801965.0, 0.0, 1.0, 118.98, 119.86, 117.81,
118.28, 31801965.0]]
```

To build a table, we can combine the column headings with the **transpose** of the row data and use the **table** function:

```
KeRF> d_data: d["dataset_data"]["data"];
KeRF> d_cols: d["dataset_data"]["column_names"];
KeRF> t: table(d_cols, transpose d_data);
KeRF> SELECT Date, Open, High, Low FROM t
```

Date	Open	High	Low
2015-12-08	117.52	118.6	116.86
2015-12-07	118.98	119.86	117.81
2015-12-04	115.29	119.25	115.11
2015-12-03	116.55	116.79	114.22
2015-12-02	117.05	118.11	116.08
2015-12-01	118.75	118.81	116.86
2015-11-30	117.99	119.41	117.75
2015-11-27	118.29	118.41	117.6
...

An alternative to using **table** would be the slightly more verbose INSERT INTO:

```
t: INSERT INTO {{{}} VALUES map(d_cols, transpose d_data)
```

Bringing everything together, we can write a simple helper function for fetching Quandl datasets and loading them into a table, permitting any kind of further querying we like:

```
def quandl_table(query) {
  url: "https://www.quandl.com/api/v3/#query"
  resp: kerf_from_json implode(`\n`, shell "curl -s "#url#" -X GET")
  cols: resp["dataset_data"]["column_names"]
  data: resp["dataset_data"]["data"]
  return table(cols, transpose data)
}
```

```
KeRF> SELECT Date, Volume FROM quandl_table "datasets/WIKI/AAPL/data.json?rows=10"
```

Date	Volume
2015-12-08	34086875.0
2015-12-07	31801965.0
2015-12-04	56351301.0
2015-12-03	40935107.0
2015-12-02	32793916.0
2015-12-01	34501246.0
2015-11-30	37074611.0
2015-11-27	13038955.0
...	...

If we load the entire dataset instead of limiting the result to the first 10 rows, we get 8824 rows and the payload is roughly 1.2mb. Our `quandl_table` routine takes somewhere around 3 seconds to run on an average laptop. Let's dig in and try to identify any performance bottlenecks by using Kerf's nanosecond-accuracy timer:

```
def itemize(f) {
  start: now()
  ticks: f({})
  timespans: map(xkeys ticks, start stamp_diff mapback ticks)
  display timespans
  display sum xvals timespans
  display timespans / (sum xvals timespans)
}

def benchmark_json(ticks) {
  d: shell "curl -s https://www.quandl.com/api/v3/datasets/WIKI/AAPL/data.json -X GET"
  ticks["curl"]: now()

  j: kerf_from_json implode(`"\n", d)
  ticks["parse"]: now()

  cols: j["dataset_data"]["column_names"]
  data: j["dataset_data"]["data"]
  t: table(cols, transpose data);
  ticks["build"]: now()

  return ticks
}
```

```
KerF> itemize(benchmark_json)
curl:5463299000, parse:3175040000, build:6035000
8644374000
curl:0.632006, parse:0.367296, build:0.000698142
```

As you can see, the time spent in **shell** and `curl` is dominant, and represents room for improvement. Parsing the JSON also takes a significant amount of time, and `INSERT INTO` is nearly instantaneous. Quandl can also emit data in a CSV format. How does this compare?

```
def benchmark_csv(ticks) {
  shell "curl -s -o t.csv https://www.quandl.com/api/v3/datasets/WIKI/AAPL/data.csv -X GET"
  ticks["curl"]: now()

  t: read_table_from_csv("t.csv", "SFFFFFFFFFFFF", 1)
  ticks["build"]: now()

  return ticks
}
```

```
KerF> itemize(benchmark_csv)
curl:7386948000, build:73101000
7460049000
curl:0.990201, build:0.009799
```

In this run, we're about 15 percent faster overall, with nearly all our time spent in **shell** and `curl`. This isn't surprising, as CSV is a much simpler file format with less structure than JSON. Clearly, if you're working with large amounts of tabular data, CSV files are more efficient.

Is there another approach? Let's try writing a dynamic library in C which extends Kerf with the IO capabilities we want. By leveraging `libcurl`, we should be able to write a simple routine which performs an HTTP GET for a specified resource and parses the result in JSON. By removing the overhead of interacting with `stdio` or the filesystem it may be possible to improve performance over using the `curl` command-line utility. For general information about dynamic libraries in Kerf, see **FFI**.

For this project we will need to work with Kerf strings. A KERF structure represents strings with a type of `KERF_CHARVEC`, and stores an *unterminated* string in the "g" field. Here's a C function which takes a Kerf string and prints out some information about it:

```
KERF string_info(KERF str) {
    printf("type was %d\n",      str->t);
    printf("length was %d\n",    (int)str->n);
    printf("first char was %c\n", str->g[0]);
    printf("string was '%s'\n",  (int)query->n, query->g);
    return 0;
}
```

For the sake of brevity, examples here will contain a minimum of error checking. They are intended to convey concepts rather than to be ideal production code. In practice when you write C functions to use with Kerf you will want to add code to verify the types of arguments you accept and print meaningful error messages if the function is invoked incorrectly.

We can produce our own strings by using `kerf_api.new_charvec`, but this copies a complete C string into a KERF structure. For more control, you can instead use `kerf_api.new_kerf` to allocate an appropriately typed and sized buffer and write into it at your convenience.

On the following page is a complete C program which can be called from Kerf to perform an HTTP GET of an arbitrary URL. For the sake of simplicity, it assumes that responses will be less than 2mb and preallocates space in a KERF structure for this response text.

Ensure that the `kerf_api.h` header file, which should be included along with the Kerf binary, is accessible, and install `libcurl` if necessary. Compile the example as follows:

```
>cc -m64 -flat_namespace -undefined suppress -dynamiclib curly.c -lcurl -o curly.dylib
```

The `-m64` option requests code be generated for a 64-bit architecture. The `-flat_namespace` and `-undefined suppress` flags are present for accessing Kerf's dylib API, as the Kerf binary itself supplies those symbols. The `-lcurl` flag instructs the linker to reference `libcurl`.

```

#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>
#include "kerf_api.h"

#define MAX_PAYLOAD 2097152 // 2mb

static size_t write_data(void *data, size_t size, size_t nmemb, void *destination) {
    KERF payload = (KERF)destination;
    size_t towrite = size*nmemb;
    // abort if the payload is larger than our preallocated buffer
    if ((payload->n) + towrite >= MAX_PAYLOAD) { return 0; }
    memcpy((payload->g) + (payload->n), data, towrite);
    payload->n += towrite;
    return towrite;
}

KERF http_get(KERF url) {
    // copy provided url into a null-terminated c string
    char* url_terminated = malloc((url->n)+1);
    strncpy(url_terminated, url->g, url->n);
    url_terminated[url->n] = '\0';

    // preallocate a fixed size buffer for the result
    KERF payload = kerf_api_new_kerf(KERF_CHARVEC, MAX_PAYLOAD);
    payload->n = 0;

    // instruct libcurl to fetch the URL and block for it
    CURL *curl_handle;
    curl_global_init(CURL_GLOBAL_ALL);
    curl_handle = curl_easy_init();
    curl_easy_setopt(curl_handle, CURLOPT_URL, url_terminated);
    curl_easy_setopt(curl_handle, CURLOPT_NOPROGRESS, 1L);
    curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, write_data);
    curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, payload);
    curl_easy_perform(curl_handle);
    curl_easy_cleanup(curl_handle);

    return payload;
}

```

After all that work, let's see how our results compare with the previous approaches:

```
http_get: dload("curly.dylib", "http_get", 1)

def benchmark_raw(ticks) {
  d: http_get("https://www.quandl.com/api/v3/datasets/WIKI/AAPL/data.json")
  ticks["curl"]: now()

  j: kerf_from_json d
  ticks["parse"]: now()

  cols: j["dataset_data"]["column_names"]
  data: j["dataset_data"]["data"]
  t: table(cols, transpose data);
  ticks["build"]: now()

  return ticks
}
```

```
KeRF> itemize(benchmark_raw)
curl:6428259000, parse:3004744000, build:5596000
9438599000
curl:0.681061, parse:0.318346, build:0.000592885
```

It's actually *slower*! Fetching data over a network will vary wildly in performance, and there simply isn't much we can control here. There's a valuable lesson here, though- by profiling before trying alternatives, we maintain a clear view of what our "optimizations" are getting us. In this case, it seems that dropping into C doesn't pay off, but now we better understand how to do it when necessary. The only way to know for sure whether this approach is appropriate for your application is to try it and take measurements.

12.5 Kerf IPC with Python

Kerf has a built-in Inter-Process Communication (IPC) protocol which is described in **Network I/O**. In this section we will look at the internal structure of this protocol as described in **The Kerf IPC Protocol** and how to interface with it using the Python programming language. Python is chosen for its general popularity, but this information will be general enough that you can apply it to any other language of your preference. Using the IPC interface we will proceed to build a simple application which uses Kerf as an in-memory database.

TCP, the Transmission Control Protocol, is the basis of most internet communication. It provides a bi-directional, fault-tolerant byte stream between a client and server. A TCP server listens for client connections on a specific *port*, which often identifies the intended purpose of a particular TCP connection. For example, HTTP traffic is generally served on port 80. Many simultaneous connections may be active on any particular port.

In the following discussion, we will build some Python functions for communicating via KIP. Explanations will use Python 3.5, the stable release version of the language at the time of writing. For information about version differences, consult <http://www.python.org>.

To open a TCP channel with a remote server, we can use the `socket` module from the standard library. Creating a connection handle with `socket.AF_INET` and `socket.SOCK_STREAM` indicates that we want an IPV4 TCP client channel. (Intuitive, no?) For easy testing purposes we will connect to `localhost`, the IP loopback device, looking for a locally-running Kerf instance which was set up to listen as a server on port 10101:

```
handle = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
handle.connect(("localhost", 10101))

# work with the connection handle here...

handle.close()
```

When reading data from the connection handle, `recv` will return however many bytes happened to be transmitted in the last TCP packet, which may or may not be the number of bytes we'd like to read. The simple solution to this clumsiness is to write a helper routine which will continue blocking and reading until we have accumulated a desired number of bytes:

```
def recv_n(handle, wanted):
    data = b""
    while(len(data) < wanted):
        data += handle.recv(wanted - len(data))
    return data
```

Now we're ready to write the meat of our program. In general, we want to JSON encode some data, build a KIP header based on this payload, transmit both, wait to read a response header, and then finally use that information to read the response payload and unpack it from JSON. From the Python standard library we'll need the `json` module for JSON encoding and decoding, the `struct` module for packing and unpacking the KIP header structure, and the `math` module for doing some of the header size calculations.

The `json.dumps` method encodes an input string as JSON. To get a byte buffer, Python 3 requires us to explicitly encode the resulting JSON string. UTF-8 is an appropriate encoding. Constructing the header requires us to calculate a shard size, which is based on the \log_2 of the payload size. The `socket.htonl` method is used to convert the wire size into network byte order (big-endian) rather than the default (on an x86 machine, little-endian).

```
code      = json.dumps([string, args]).encode("UTF-8")
shard_size = int(math.ceil(math.log(len(code)+16, 2)))
wire_size  = int(2**shard_size)
wire_net   = socket.htonl(wire_size)
padding    = wire_size - (len(code)+16)
```

The `struct.pack` method provides a convenient way to assemble all our various-sized fields together into a byte buffer. The format string provided uses `x` to indicate padding bytes, `b` to indicate a signed byte and `I` to indicate an unsigned 32-bit integer. Hardcoding execution, response and display types, we can arrive at something like the following to send the header, payload and padding:

```
packing = "xxxxbbbxixxxIbxxbIIxxxx"
packed = struct.pack(packing, 4, 1, 0, wire_net, shard_size, -1, 1, len(code))
handle.send(packed)
handle.send(code)
handle.send(bytes(padding * [0]))
```

Now we wait for a response and unpack the result header. A KIP header is precisely 32 bytes, so we perform a blocking read using our `recv_n` helper method. `struct.unpack` is an inversion of `struct.pack`, and can share the same format string. The only data we actually need from the header is the payload size and the wire size, which allow us to calculate how many padding bytes should be discarded. `socket.ntohl` reverses `socket.htonl`, converting network byte order into a usable native byte order.

```
data      = recv_n(handle, 32)
header    = struct.unpack(packing, data)
totality  = 16 + socket.ntohl(header[3]) # wire size + 16
size      = header[7]                   # payload size
payload   = recv_n(handle, size)
recv_n(handle, totality - (size + 32))
```

All that remains is to decode our payload bytes as UTF-8 and decode the JSON into a useful result:

```
json.loads(payload.decode("UTF-8"))
```

All together now, kerf_ipc.py:

```
import socket, json, struct, math

def recv_n(handle, wanted):
    # block for n bytes
    data = b""
    while(len(data) < wanted):
        data += handle.recv(wanted - len(data))
    return data

def sync_send(handle, string, *args):
    # calculate header information
    code = json.dumps([string, args]).encode("UTF-8")
    shard_size = int(math.ceil(math.log(len(code)+16, 2)))
    wire_size = int(2**shard_size)
    wire_net = socket.htonl(wire_size)
    padding = wire_size - (len(code)+16)

    # send the header, payload and padding
    packing = "xxxxbbbxixxxlxbxblxxxx"
    packed = struct.pack(packing, 4, 1, 0, wire_net, shard_size, -1, 1, len(code))
    handle.send(packed)
    handle.send(code)
    handle.send(bytes(padding * [0]))

    # parse out the response
    data = recv_n(handle, 32)
    header = struct.unpack(packing, data)
    totality = 16 + socket.ntohl(header[3]) # wire size + 16
    size = header[7] # payload size
    payload = recv_n(handle, size)
    recv_n(handle, totality - (size + 32))

    return json.loads(payload.decode("UTF-8"))
```

Let's see a simple example of these routines in action. First, start up a Kerf instance waiting for IPC traffic:

```
> ./kerf -q -p 10101
KeRF>
```

Then run a little Python script:

```
import socket, kerf_ipc
handle = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
handle.connect(("localhost", 10101))
print(kerf_ipc.sync_send(handle, "range 2+3"))
handle.close()
```

```
> python3.5 demo.py
[0, 1, 2, 3, 4]
>
```

Two entirely different languages cooperating hand in hand- beautiful!

12.5.1 HudsucKerf By Proxy

With these new abilities in hand, one nifty thing we can do is use Python to construct user interfaces for Kerf applications. Imagine describing a simple product and inventory tracking system in Kerf (call it hudsuc.kerf):

```
widgets: {{id, name, price, weight}}
orders:  {{id, widget_id, quantity, time}}

def new_widget(name, price, weight) {
    INSERT INTO widgets VALUES (count(widgets), name, price, weight);
}

def get_widgets() {
    return SELECT * FROM widgets;
}
```

We can interact with this database directly via the Kerf REPL, but perhaps we want fancier custom displays and menus. It's possible to do this directly with Kerf and dynamic libraries, as we described in some earlier examples. Alternatively, maybe using Python is easier for your application:

```
import socket, kerf_ipc
db = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
db.connect(("localhost", 10101))

def c_quit():
    db.close()
    exit(0)

def c_idea():
    name  = input("product name: ")
    price = input("price:       ")
    weight = input("weight:      ")
    kerf_ipc.sync_send(db, "new_widget($1, $2, $3)", name, price, weight)
    print("great idea, boss!")

def c_widgets():
    w = kerf_ipc.sync_send(db, "get_widgets()")
    print("price\tname")
    print("-----\t-----")
    for row in range(len(w["name"])):
        print(w["price"][row] + "\t"+w["name"][row])

commands = {"quit": c_quit, "widgets": c_widgets, "idea": c_idea}

print("Idea Management Gizmo v0.1")
print("(c) 1959 HudsucKerf Industries")
while True:
    commands[input("> ")]()
```

Notice how easily we can call Kerf procedures from Python (and how much more pleasant they are to write than SQL stored procedures!), separating database storage and reporting logic from the frontend application. If your program needs persistent storage, it would be a simple matter to make Kerf use memory-mapped IO to automatically serialize tables to disk.

Let's see a quick command-line session:

```
> kerf -q -p 10101 hudsuc.kerf
KeRF> widgets
```

id	name	price	weight

```
KeRF>
server: new connection from ::ffff:127.0.0.1 on socket 6
```

```
> python3.5 hudsuc.py
Inventory Management Gizmo v0.1
(c) 1959 HudsucKerf Industries
> idea
product name: extruded plastic dingus
price:        1.79
weight:       2.5
great idea, boss!
> idea
product name: bendy straw
price:        0.05
weight:       0.01
great idea, boss!
> widgets
price          name
-----
1.79           extruded plastic dingus
0.05           bendy straw
> quit
```

Obviously, this is just scratching the surface of what becomes possible. You could build graphical UIs, service many concurrent users, interface with or expose web APIs, and much more.