CJam Tutorial and Docs Documentation

Release 1.0

???

Jul 15, 2017

Contents

CHAPTER 1

Tutorials

Tutorial #1: Introduction to CJam

Getting started

There are currently two ways to run CJam code.

The easy way is to simply use the online CJam interpreter. This will be good for most purposes, but is not recommended if you need code to run quickly or the code generates copious amounts of output.

Alternatively, if you have Java, you can download the JAR from SourceForge and run like

```
java -jar cjam-someversion.jar file.cjam
```

You can also launch an interactive shell with

```
java -cp cjam-someversion.jar net.aditsu.cjam.Shell
```

Hello, World!

CJam's main source of memory is a stack which can hold values. You can push elements onto the stack and modify the stack with operators. At the end of the program, the contents of the stack are automatically printed.

`Hello, World!` is very easy in CJam – we can just push a string onto the stack and let automatic printing do its job (permalink).

```
"Hello, World!"
```

Simple!

Numerical calculations in CJam

To perform calculations in CJam, we can push integers onto the stack and apply operators to them. For example, the program

```
7 8 +
```

results in the output (permalink)

```
15
```

by first pushing `7` and `8` onto the stack, then performing the `+` operator, which adds two integers.

Another example is `4 2 3 * 7 - +`, which results in `3` (permalink). Here is a trace:

```
Instruction       Function              Stack
-----------       ------------          -----
4                 Push 4                [4]
2                 Push 2                [4 2]
3                 Push 3                [4 2 3]
*                 Multiply top two      [4 6]
7                 Push 7                [4 6 7]
-                 Subtract top two      [4 -1]
+                 Add top two           [3]
```

For debugging, you can insert the two-char operator `ed` to print the current stack. For example, `4 2 3 * 7 ed - +` (permalink) prints the state of the stack after the `7` is pushed.

In addition to integers, CJam also has doubles. For example, the program `1.3 2.6 +` results in `3.9000000000000004` (permalink), which is close enough to the expected `3.9` (silly floating point numbers!).

Here are examples of operators which do numerical calculations:

```
+ - * /          Add, subtract, multiply and divide respectively
( )              Decrement and increment by one respectively
%                Modulo
#                Exponentiate/power
& | ^ ~          Bitwise AND, OR, XOR and NOT respectively (for integers)
```

Note that division between two integers in CJam results in integer division, which keeps the integer part of the result. For example, `5 2 /` results in `2`, rather than `2.5` (permalink). To get floating point division, one of the arguments needs to be a double, for instance both `5.0 2 /` and `5 2. /` result in `2.5` (permalink).

Blocks and variables

Block is a data type in CJam which represents a code block. They can be executed with the `~` operator, e.g. the program

```
6 {7*} ~
```

gives the output (permalink)

```
42
```

You may notice that `~` executes a block when used on blocks, but performs bitwise not when used on integers. CJam operators are very heavily overloaded, with the same operator doing different things depending on what is at the top of the stack.

Blocks in CJam are first class objects. They can be assigned to variables, allowing them to act as functions. CJam has 26 variables, one for each uppercase letter, and a variable is assigned to via `:<variable>`. For example, the program

```
{7*}:F 6 F
```

results in (permalink)

```
{7*}42
```

Note how the block is also in the output. When you assign something to a variable, it is not popped from the stack, so here the block gets outputted along with the `42` when the stack is automatically printed.

Of course, you can also assign numbers and strings to variables, which just get pushed onto the stack when "used". For instance,

```
"HOUSE":A "BOAT":B A B B B A
```

gives the output (permalink)

```
HOUSEBOATHOUSEBOATBOATBOATHOUSE
```

Each of CJam's variables is preinitialised with a value, the full list of which can be found here.

Arrays and characters

An array is just a collection of items, for instance

```
[1 2 3 "foo"]
```

is an array of four things – three integers and a string. You can get the length of an array with the `,` operator, so `[1 2 3 "foo"],` would result in `4` (permalink).

Funnily enough, there is no special literal for general arrays – although arrays can be formed by surrounding the items with square brackets `[]`, in actuality `[` and `]` are both *operators* which start and end an array respectively. This lets you do things like

```
1 2 3 "foo"]
```

which also creates an array of the same four items, but instead works because the `]` here wraps the *whole stack* into an array.

In CJam, strings are actually a special case of arrays – CJam strings are just arrays of characters. Aside from integers, doubles, blocks and arrays, the character is CJam's fifth and final data type. Character literals take the form `'<character>`, i.e. a single quote/apostrophe followed by the character. For example `5 '$` prints `5$` (permalink).

There is no escape character for characters, so `''` is actually just the single quote character. There *is*, however, an escape character for strings – the backslash `\`, like most major languages. But unlike most languages, backslashes only escape double quotes and other backslashes in CJam, with every other character preserved as-is. In particular, newlines are perfectly okay in strings, e.g.

```
"This is a double quote: \"
Backslash followed by 'n' is not special: \n"
```

results in (permalink)

```
This is a double quote: "
Backslash followed by 'n' is not special: \n
```

Converting types

Now that we've seen examples of each CJam data type, here is a summary of operators which convert between types:

```
`       String representation
c       Convert to character
d       Convert to double
i       Convert to integer
s       Convert to string
~       Evaluate string/block
```

For example, `33 c` converts 33 to its respective ASCII character, `'!`, and `'! i` gives us back 33. Some combinations don't make sense and give an error, such as converting a string to an integer. However, counterintuitively, it's possible to convert a non-empty string to a character to give the first char (permalink):

```
"abcde" c    ->    'a
```

Revisiting the `5 2 /` example from earlier, if we already had 5 and 2 on the stack and wanted float division, this means that we can perform `d` to convert the 2 to a double before dividing (permalink):

```
5 2 d /      ->    2.5
```

Another note is that backtick `` ` `` and `s` differ primarily in how arrays are turned into strings. For example, `[1 2 3] ` ` results in the string `"[1 2 3]"` while `[1 2 3] s` results in `"123"`.

Input and output

In CJam, there are three operators for getting input via STDIN:

```
l       Read line
q       Read all input
r       Read token (whitespace-separated)
```

For instance,

```
l i 2 *
```

is a simple program which doubles an input integer (permalink). `ri2*` or `qi2*` would work just as well here (whitespace is typically unnecessary except between numeric literals).

There is also a fourth operator, `ea`, which pushes arguments from command-line onto the stack as an array. This feature is only available in the offline interpreter.

As for output, aside from the automatic printing upon program termination, CJam also has two specific operators for output:

```
o       Print value
p       Print string representation and newline
```

Stack manipulation

Much of CJam's power as a golfing language comes from its stack manipulation operators, which reduce the need to assign to variables. Here are some of these operators

```
Operator      Function                    Stack after [1 2 3 4]
--------      --------                    ---------------------
<n> $         Copy top nth from stack     [1 2 3 4 3] (for 1$)
                                          [1 2 3 4 2] (for 2$)
;             Pop and discard             [1 2 3]
\             Swap top two                [1 2 4 3]
@             Rotate top three            [1 3 4 2]
_             Duplicate                   [1 2 3 4 4]
```

Note that `$` starts counting from `0`, so `0$` is the same as `_`. It also allows for negative numbers, e.g. `-1$` copies the bottom of the stack.

The easy way to remember which way `@` rotates is that since `\` moves the second element from the top to the top, it's more useful for `@` to move the third element from the top to the top.

Example program: Distance calculator

Let's take a look at how we might write a program which takes in 4 numbers `x1 y1 x2 y2` and outputs the Euclidean distance between `(x1, y1)` and `(x2, y2)`, i.e.

```
sqrt((x1 - x2)^2 + (y1 - y2)^2)
```

For example, the input `3 7 4 5` would output `sqrt(5) ~ 2.23`.

First, we need to read in and convert the input, for which we can use `l~`. `~` evaluates the whole string, leaving the stack like

```
[x1 y1 x2 y2]
```

We can then move `y1` to the top with `@`, bringing the `y`s together, after which we subtract with `-` and square with `_*` (duplicate and multiply). This gives:

```
[x1 x2 (y2-y1)^2]
```

As a side comment, although it appears it could, `2#` (power of 2) can't be used in place of `_*` here due to the preceding `-`, which together is parsed as `-2 #` (power of -2). Adding a space in between like `- 2#` would work, but it is better to use the `m` operator, which is designed to act as subtraction when followed by a numeric literal (i.e. `m2#` can be used instead of `-_*`).

Moving on, we can move the top of the stack to the bottom by rotating twice with `@@`, giving:

```
[(y2-y1)^2 x1 x2]
```

Then we subtract and square again with `-_*`:

```
[(y2-y1)^2 (x1-x2)^2]
```

Finally, we add and square root with `+.5#`, thus leaving the final stack as

```
[((y2-y1)^2 + (x1-x2)^2)^.5]
```

Altogether, this gives the program (permalink)

```
l~@-_*@@-_*+.5#
```

Considering we only used the basic operators here, this is already a fairly short program. However, CJam actually has a builtin hypotenuse function `mh`, which is another two-char operator like `ed` (`e` is for *extended* operators, while `m` is for *mathematical* operators). Using this, we can make the program even shorter with (permalink):

```
l~@-@@-mh
```

Tutorial #2: Basic array manipulations

Now that we've had a quick introduction, we'll be taking a look at the core features which make CJam such an expressive language, starting with the ever-versatile array.

Wrapping and unwrapping

We've seen that arrays can be created with `[` and `]`, which are actually operators. Additionally, there is an `a` command, which wraps the top element of the stack into an array (permalink).

```
2 a     ->     [2]
```

Arrays can be unwrapped using `~`, pushing its elements onto the stack (permalink):

```
[1 2 3] ~     ->     1 2 3
```

Note that this doesn't work for strings, i.e. arrays consisting of only characters, for which `~` is eval. If dumping a string's characters to the stack is desired, then an empty for-each loop `{}/` can be used instead (we'll look more closely at blocks in a later tutorial).

Concatenation

Two arrays can be concatenated with `+` (permalink):

```
[1 4 2] [8 5 7] +     ->     [1 4 2 8 5 7]
"house" "boat"  +     ->     "houseboat"
```

Concatenating a non-array element is equivalent to concatenating a one-element array, negating the need to use `a` (permalink):

```
[1 2 3] 4  +     ->     [1 2 3 4]
[1 2 3] 4a +     ->     [1 2 3 4]
4 [1 2 3]  +     ->     [4 1 2 3]
4a [1 2 3] +     ->     [4 1 2 3]
```

However, it is important to remember than strings are arrays, or else there might be some unexpected results (permalink):

```
[1 2 3] "four"  +     ->     [1 2 3 'f 'o 'u 'r]
[1 2 3] "four"a +     ->     [1 2 3 "four"]
```

Repetition

Given an integer `n` and an array, `*` repeats the array `n` times (permalink):

```
[1 2 3] 5 *     ->     [1 2 3 1 2 3 1 2 3 1 2 3 1 2 3]
"ha"    2 *     ->     "haha"
```

Array length and ranges

Recall that `,` on an array gives the length of an array, so

```
["this" "array" "has" "five" "elements"] ,
```

results in `5` (permalink). `,` on an integer *creates* an array, namely it is a range operator which produces `[0 1 ... n-1]`: (permalink):

```
5,     ->     [0 1 2 3 4]
```

`,` also acts as a range operator on characters, where it creates a string instead. For instance, `'a,` results in the first 97 ASCII characters, including the initial unprintables (permalink).

Getting and setting

Getting an element at a specific index of an array can be done with =. A nice feature is that out-of-bounds indices wrap around. (permalink)

```
[1 4 2 8 5 7]      0 =       ->    1
[1 4 2 8 5 7]      5 =       ->    7
[1 4 2 8 5 7]      6 =       ->    1
[1 4 2 8 5 7]     -2 =       ->    5
[1 4 2 8 5 7] -1000 =       ->    2
```

= can also take a block, returning the first element for which the given condition is true. For example, the following returns the first positive integer in the array (permalink):

```
[0 -1 2 -3 0 4] {0>} =     ->     2
```

Setting a position in an array is done with `t`, with the index followed by the element (permalink):

```
[1 4 2 8 5 7]  0 -3 t       ->    [-3 4 2 8 5 7]
```

Similarly to =, `t` also wraps around for out-of-bounds indices. However, unlike =, `t` doesn't take a block.

Slices

Array slicing is done with > and < (permalink):

```
[1 4 2 8 5 7]  2 <      ->     [1 4]
[1 4 2 8 5 7]  2 >      ->     [2 8 5 7]
[1 4 2 8 5 7] -3 <      ->     [1 4 2]
[1 4 2 8 5 7] -3 >      ->     [8 5 7]
```

For example, one way of getting the lowercase letters is `'{,97>` (permalink).

Getting every nth element can be done with `%` (permalink):

```
[1 4 2 8 5 7]  2 %    ->    [1 2 5]
[1 4 2 8 5 7]  3 %    ->    [1 8]
[1 4 2 8 5 7]  W %    ->    [7 5 8 2 4 1]
[1 4 2 8 5 7] -2 %    ->    [7 8 4]
```

For negative numbers the step is backwards, so `W%` is a common idiom for reversing an array, as `W` is preinitialised to -1.

Uncons

Sometimes you want to just pop from an array, removing the first or last element while leaving the rest of the array intact. This can be done with `(` and `)` respectively (permalink):

```
[1 4 2 8 5 7]  (    ->    [4 2 8 5 7] 1
[1 4 2 8 5 7]  )    ->    [1 4 2 8 5] 7
```

Sort

Sorting can be done with `$` (permalink):

```
[1 4 2 8 5 7]  $    ->    [1 2 4 5 7 8]
```

If two elements aren't comparable, an error is sensibly thrown.

`$` can optionally take an additional block argument which determines what key to sort by. For example, `$` on its own sorts strings by ASCII values, while `{el} $` sorts strings by their lowercase counterparts, giving a case-insensitive search. Compare (permalink):

```
["Bee" "candy" "Cake" "apple"]      $   ->   ["Bee" "Cake" "apple" "candy"]
["Bee" "candy" "Cake" "apple"] {el} $   ->   ["apple" "Bee" "Cake" "candy"]
```

`el` and `eu` convert strings to lowercase and uppercase respectively.

Another example is `{3-z} $`, which sorts an array of numbers by the key `|n-3|`, with `z` being the absolute value operator for numbers (permalink):

```
[-2 4 2 6 9 1 3 -1] {3-z} $ p    ->    [3 4 2 1 6 -1 -2 9]
```

Note how 4 comes before 2 in the resulting array, even though they are both the same distance from 3. CJam uses Java's `Collections.sort`, which is stable and does not reorder equal elements.

Find index

If both top elements of the stack are arrays, `#` returns the first index of a subarray within another, or `-1` if the subarray is not found (permalink):

```
[1 0 1 1 0 0 1 0 1 1 0] [1 1 0] #    ->    2
[1 0 1 1 0 0 1 0 1 1 0] [0 1 0] #    ->    5
[1 0 1 1 0 0 1 0 1 1 0] [0 0 0] #    ->    -1
```

It is important to remember that strings are arrays too. For instance, the second example below won't work because `["fish"]` is not a subarray of `"one fish two fish" = ['o 'n 'e ...]` (permalink).

```
"one fish two fish" "fish"  #    ->    4
"one fish two fish" "fish"a #    ->    -1

["one" "fish" "two" "fish"] "fish"  #    ->    -1
["one" "fish" "two" "fish"] "fish"a #    ->    1
```

If one of the elements is an array but the other is a number/character, `#` returns the first index of the number/character in the array, or `-1` if not found (permalink):

```
[1 4 2 8 5 7] 1 #    ->    0
[1 4 2 8 5 7] 5 #    ->    4
[1 4 2 8 5 7] 9 #    ->    -1
"mhsjmdkmgc" 'j #    ->    3
"mhsjmdkmgc" 'q #    ->    -1
```

Like `=` and `$`, `#` can also take a block, returning the index of the first element which satisfies the given condition. For example, `{0>} #` gives the index of the first positive element (permalink):

```
[-3 -2 0 4 9 -3 0 2] {0>} #    ->    3
```

Split

`/` splits an array based on another. The most common use for this is to split a string, e.g. `S/` splits by spaces and `N/` splits by newlines (permalink):

```
"one fish two fish"       S /    ->    ["one" "fish" "two" "fish"]
"one fish two fish"  "fish" /    ->    ["one " " two " ""]
"one fish two fish" "uh-oh" /    ->    ["one fish two fish"]
```

In fact, `/` works for any two general arrays (permalink):

```
[1 4 "cake" 3 "blue" 4 "cake" 2] [4 "cake"] /    ->    [[1] [3 "blue"] [2]]
```

Similarly, `%` can also be used for splitting, but removes empty chunks in the resulting array (permalink):

```
"one fish two fish" "fish" % p    ->    ["one " " two "]
```

`/` also works if one element is an array and the other is a number. In this situation the array is split into chunks of size equal to the number, except possibly the last chunk if there aren't enough elements (permalink):

```
[1 4 2 8 5 7] 2/    ->    [[1 4] [2 8] [5 7]]
[1 4 2 8 5 7] 3/    ->    [[1 4 2] [8 5 7]]
[1 4 2 8 5 7] 4/    ->    [[1 4 2 8] [5 7]]
```

Join

`*` takes two arrays – the second array is riffled between the elements of the first. For example, `", " *` joins an array of values with commas (permalink):

```
[1 4 2 8 5 7] ", " *    ->    1, 4, 2, 8, 5, 7
```

The actual resulting array looks like this:

```
[1 ', '  4 ', '  2 ', '  8 ', '  5 ', '  7]
```

We can join with any array, e.g. (permalink):

```
[1 7 2 9] ["a" 0] *    ->    [1 "a" 0 7 "a" 0 2 "a" 0 9]
```

Joining can be combined with splitting to give an idiom for search-and-replace (permalink):

```
Replace "13" with "..":
"12313132131231" "13"/ ".."*    ->    "123....2..1231"

Collapse runs of spaces:
"a   b c    d  " S% S* p        ->    "a b c d"
```

Set operations

CJam arrays can actually be used as sets. Like sorting, CJam set operations are stable, preserving element order.

The set operations are:

```
&    Set AND, or intersection
|    Set OR, or union
^    Set XOR, or symmetric difference
-    Set minus, or relative complement
```

For instance (permalink):

```
[3 1 4 1 5 9] [2 6 5 3] &    ->    [3 5]
[3 1 4 1 5 9] [2 6 5 3] |    ->    [3 1 4 5 9 2 6]
[3 1 4 1 5 9] [2 6 5 3] ^    ->    [1 4 9 2 6]
[3 1 4 1 5 9] [2 6 5 3] -    ->    [1 4 1 9]
```

A common idiom for removing duplicates from an array is to simply do `L|`, finding the union of an array with the empty list.

Example program: Name sorter

Suppose we have a list of properly capitalised names, e.g.

```
John Doe
Jane Smith
John Smith
Tommy Atkins
Jane Doe
Foo Bar
```

Let's say we want to sort this list by last name, with ties broken by first name.

First we need to read in the input with `q` and split by newlines with `N/`. This gives an array of strings:

```
["John Doe" "Jane Smith" "John Smith" "Tommy Atkins" "Jane Doe" "Foo Bar"]
```

Sorting normally is clearly not sufficient here, so we'll need to sort with a block like `{}$`. To include a tie breaker in our sort, we will use an array as the key. Since last names take priority, the key array will simply be the last name followed by the first name.

Suppose we're in the situation where we're looking at a name, say `"John Doe"`. To get the key we just split by spaces with `S/` to get `["John" "Doe"]`, then reverse the array with `W%` to give the sorting key `["Doe" "John"]`.

Putting this together, after `{S/W%}$` we get

```
["Tommy Atkins" "Foo Bar" "Jane Doe" "John Doe" "Jane Smith" "John Smith"]
```

All that remains is to join with newlines `N*` to get the final list:

```
Tommy Atkins
Foo Bar
Jane Doe
John Doe
Jane Smith
John Smith
```

And the final program is (permalink):

```
qN/{S/W%}$N*
```

Now what if we have middle names, like `John B. Doe`? Our current program would still work, but our sorting key would prioritise middle names before first names. If we want to prioritise first names *before* middle names, then we would need to change the key a little.

Splitting `"John B. Doe"` by spaces gives `["John" "B." "Doe"]`, but the desired priority order is `["Doe" "John" "B."]`. We simply need to move the last name to the front, and to do so we start by popping the last name off:

```
["John" "B." "Doe"] )       ->     ["John" "B."] "Doe"
```

Then we swap `\` and concatenate `+`, right? Almost...

```
["John" "B." "Doe"] )\+     ->     ['D 'o 'e "John" "B."]
```

Because `"Doe"` is a string, i.e. an array of characters, concatenating the two arrays dumps the *characters* of `"Doe"` at the front, rather than adding on the string as a whole piece. To fix this, we just need to add an `a` to wrap the last name in an array, giving `)a\+` (permalink):

```
["John" "B." "Doe"] )a\+    ->     ["Doe" "John" "B."]
```

Using this new key gives the program (permalink):

```
qN/{S/)a\+}$N*
```

CHAPTER 2

Docs

New Operators in CJam 0.6.5

Vector operators

`.` can be applied before a binary operation to vectorise it so that it performs operations element-wise.

```
[1 2 3] [7 6 2] .+   ->  [8 8 5]
[1 2 3] [7 6 2] .*   ->  [7 12 6]
[1 2 3] [7 6 2] .#   ->  [1 64 9]
[1 2 3] [7 6 2] .e<  ->  [1 2 2]
[1 2 3] [7 6 2] .\   ->  [7 1 6 2 2 3]
```

Since `.` is just a modifier, it can be chained.

```
[[1 2 3] [7 6 2]] [[8 9 1] [2 0 3]] ..+  ->  [[9 11 4] [9 6 5]]
```

`w` – while loops

Takes two blocks – a condition followed by a loop body – and performs a while loop. The condition is tested for before each loop iteration, and is consumed like `g`.

For example,

```
1 {_5<} {_'X*N+o)} w ;
```

prints a triangle of `X`s by incrementing a number until it is no longer less than 5.

```
X
XX
XXX
XXXX
```

mQ – integer square root

Take an integer or double and finds the square root, truncated to int.

```
99 mQ     ->  9
100 mQ    ->  10
3.14 mQ   ->  1
```

mR – random choice

Given an array, chooses a (pseudo-)random element, for example:

```
[1 2 3 4] mR  ->  2
```

e# – line comment

Simply comments out the rest of the line.

```
T1{_2$+}A*]S* e# Fibonacci
```

e% – string formatting a la printf

Takes a format string and an element/array and performs string formatting using printf.

```
"(%d, %d)" [1 2] e%  ->  (1, 2)
"%.08f" 3.14 e%      ->  3.14000000
```

e* – repeat each item

Given an array `A` and a number `n`, returns a new array which consists of every element of `A` repeated `n` times.

```
[1 2 3] 5 e*         ->  [1 1 1 1 1 2 2 2 2 2 3 3 3 3 3]
["abc" "def"] 3 e*   ->  ["abc" "abc" "abc" "def" "def" "def"]
```

e\ – swap array items

Given an array and two indices, swaps two elements of the array at the specified indices.

```
[0 1 2 3 4] 1 4 e\   ->  [0 4 2 3 1]
```

e= – count occurrences

Given an array and an element, counts the number of times the element appears in the array.

```
[0 0 1 0 2 2] 0 e=                  ->  3
[[0 1] [1 0] [1 1] [0 1]] [0 1] e=  ->  2
```

e! – permutations

Given an array, returns an array consisting of **all unique** permutations, in lexicographical order.

```
[3 1 2] e!  ->  [[1 2 3] [1 3 2] [2 1 3] [2 3 1] [3 1 2] [3 2 1]]
"BAA" e!    ->  ["AAB" "ABA" "BAA"]
```

If given a number, an implicit call to , (range) is performed first.

```
3 e!  ->  [[0 1 2] [0 2 1] [1 0 2] [1 2 0] [2 0 1] [2 1 0]]
```

m! – factorial and permutations with duplicates

Given an array, returns an array consisting of **all possible** permutations. Unlike e!, this is formed by applying each permutation in lexicographical order to the array, and hence the result may not necessarily be sorted.

```
[3 1 2] m!  ->  [[3 1 2] [3 2 1] [1 3 2] [1 2 3] [2 3 1] [2 1 3]]
"BAA" m!    ->  ["BAA" "BAA" "ABA" "AAB" "ABA" "AAB"]
```

Given a number, finds the factorial of the number. Note that the number is truncated, so this is not equivalent to the gamma function.

```
5 m!     ->  120
3.14 m!  ->  6
```

e_ – flatten

Flattens an array completely, no matter how nested it is.

```
[[1 2 3] [[4] [[5] [6]]]] e_  ->  [1 2 3 4 5 6]
```

e` and e~ – run length encode/decode

e` encodes a sequence using run-length encoding, which keeps counts of the number of times the same element appears in a row.

```
"AAAABCCCAAC" e`                    ->  [[4 'A] [1 'B] [3 'C] [2 'A] [1 'C]]
[[1 0] [1 0] [1 1] [1 1] [1 1]] e`  ->  [[2 [1 0]] [3 [1 1]]]
```

e~ does the opposite, and decodes a run-length encoding.

```
[[4 'A] [1 'B] [3 'C] [2 'A] [1 'C]] e~  ->  "AAAABCCCAAC"
[[2 [1 0]] [3 [1 1]]] e~                ->  [[1 0] [1 0] [1 1] [1 1] [1 1]]
```

ew – overlapping slices

Given an array and a slice length, returns all overlapping slices of the array with the given length.

```
[1 2 3 4 5 6] 3 ew  ->  [[1 2 3] [2 3 4] [3 4 5] [4 5 6]]
```

`e[` and `e]` – left/right array padding

Pads an array to a given length with a specified element.

```
"abc" 10 '0 e[    ->  "0000000abc"
[1 2 3 4] 7 0 e[  ->  [0 0 0 1 2 3 4]

"abc" 10 '0 e]    ->  "abc0000000"
[1 2 3 4] 7 0 e]  ->  [1 2 3 4 0 0 0]
```

`m*` with array and number – Cartesian power

Given an array and a number, returns the Cartesian product of the array repeated the given number of times. Alternatively, this is all arrays of the given length consisting of elements from the specified array. Elements can be repeated.

```
[0 1] 3 m*  ->  [[0 0 0] [0 0 1] [0 1 0] [0 1 1] [1 0 0] [1 0 1] [1 1 0] [1 1 1]]
"AAB" 2 m*  ->  ["AA" "AA" "AB" "AA" "AA" "AB" "BA" "BA" "BB"]
```

`ee` – enumerate

Given an array, returns an array of `[index element]` pairs.

```
"ABCDEFG" ee  ->  [[0 'A] [1 'B] [2 'C] [3 'D] [4 'E] [5 'F] [6 'G]]
```

`&` and `|` for blocks – "if" with only "then" or only "else"

`&` checks if a condition is truthy, and if so executes the block.

```
100 0 {3+} &  ->  100
100 1 {3+} &  ->  103
```

`|` checks if a condition is falsy, and if so executes the block.

```
100 0 {3+} |  ->  103
100 1 {3+} |  ->  100
```

Operator summary

Notes

- If an operator takes arguments of different types then usually the arguments may be given in a different order to the one listed.
- Most block operations which require an array will implicitly call `,` (range) when given a number. Exceptions include `:`, `.` and `f`.

Op.	Arguments	Description	Example
!	`Any !`	Boolean "not"	`5 ! -> 0`
#	`x:Num y:Num #`	Power/exponentiation, `x^y`	`2 .5 # -> 1.4142135623730951`
	`Array Any #`	Find index, or -1 if not present	`"banana" "na" # -> 2`
$	`Int $`	Copy from stack	`-4 "ab" 3 1$ -> -4 "ab" 3 "ab"`
	`Array $`	Sort	`[3 0 2 1] $ -> [0 1 2 3]`
%	`x:Num y:Num %`	Modulo, `x mod y`	`27 5 % -> 2`
	`Array Num %`	Every nth	`[0 1 2 3 4] 2 % -> [0 2 4]`
	`Array Array/Char %`	Split by	`"1,2,3" ', % -> ["1" "2" "3"]`
	`Array Block %`	Map	`[1 2 3] {2*3+} % -> [5 7 9]`
&	`Int/Char Int &`	Bitwise "and"	`5 6 & -> 4`
	`Array Any &`	Setwise intersection	`[6 1 8 1] [6 1 1 2] & -> [6 1]`
(	`Num/Char (`	Decrement	`42 ( -> 41`
	`Array (`	Uncons from left	`[2 7 1 8] ( -> [7 1 8] 2`
)	`Num/Char )`	Increment	`42 ) -> 43`
	`Array )`	Uncons from right	`[2 7 1 8] ) -> [2 7 1] 8`

Operator details

Basic arithmetic

`x:Num y:Num #` – power/exponentiation

Finds `x` raised to the power of `y`.

```
5 3 #   ->  125
2 .5 #  ->  1.4142135623730951
0 0 #   ->  1
```

Logical operations

`Any !` – boolean "not"

Return 0 if the element is truthy, otherwise 1.

```
5 !       ->  0
"abc" !   ->  0
[1 0] !   ->  0
0 !       ->  1
"" !      ->  1
```

Tip: `! !` can be used to give the truthiness of an element.

Pages marked with * are incomplete and under construction.