prelude.ls

Overview | Installation

map | filter | reject | partition

find | each | head | tail

last | initial | empty | values

keys | length | cons | append

join | reverse | fold | fold1

foldr | foldr1 | unfold | andList

orList | any | all | unique

sort | sortBy | compare | sum

product | mean | concat | concatMap

maximum | minimum | scan | scan1

scanr | scanr1 | replicate | take

drop | splitAt | takeWhile | dropWhile

span | breakIt | listToObj | objToFunc

zip | zipWith | zipAll | zipAllWith

compose | curry | id | flip

fix | lines | unlines | words

unwords | max | min | negate

abs | signum | quot | rem

div | mod | recip | pi

tau | exp | sqrt | ln

pow | sin | cos | tan

asin | acos | atan | atan2

truncate | round | ceiling | floor

isItNaN | even | odd | gcd

lcm

is the

It is based in

b

a partially

```
//=> 'hell'
```

bject, and a

```
*2; }, {d: 3,
```

can also use

```
map {on: 1, off: 0}, <[ on on off on of
f ]>
#=> [1, 1, 0, 1, 0]
```

```
reject([fals , tru ], {a: 0, b: 1, c: 0
});
//=> {a: 0, c: 0}
```

Installation

Install via npm: `npm install prelude-ls` and then you can require it as a standard node module.

For the browser, just include the `prelude-browser-min.js` file. It will install itself as `prelude` under the global namespace.

To import all prelude.ls functions to somewhere, you can use `prelude.installPrelude(target)`. For instance, calling `prelude.installPrelude(window)` in the browser will install all the functions as global variables. In node, you can do `require('prelude-ls').installPrelude(global);`. In LiveScript you can simply do `import prelude` in the browser, and `global <<< require \prelude-ls` in node.

Function Reference

Examples of usage in LiveScript on the left, examples using JavaScript on the right. The examples may not be related. If you want to play around with prelude.ls, take a look at the LiveScript site, it has the library preloaded for use in the compiler. In LiveScript, you may call any camel cased functions using hyphens, eg. you may use `concat-map` instead of `concatMap`.

All functions except those that take a variable number of arguments are 'curried' - if called with less than the specified number of arguments, they will return a partially applied function. Eg. `takeThree = take 3` and then `takeThree [1 to 10] #=> [1, 2, 3]`.

Iterating over a string only works on IE8+. "Callable" refers to either a

function, or an object (anything with properties). A callable that is a function takes only one argument.

map

Callable -> List -> List
Callable -> Object -> Object
Callable -> String -> String

Applies the supplied callable to each item in the collection, returns a new collection with the results. The length of the result is the same as the length of the input.

LiveScript

```
map (* 2), [1 to 5] #=> [2, 4, 6, 8, 10]
map (.toUpperCase!), 'haha' #=> 'HAHA'
map (+ 2), {a: 2, b: 3, c: 4} #=> {a: 4, b: 5, c: 6}
map {one: 1, two: 2}, <[ one two ]> #=> [1, 2]
map [\off, \on], {power: 1, light: 0} #=> {power: 'on', light: 'off'}
map (.n), [n 3; n 1] #=> [3, 1]
```

JavaScript

```
map (function (x){ return x * 2; }, [1, 2, 3, 4, 5]);
//=> [2, 4, 6, 8, 10]
map (someFunction , aList ); //=> a list of something
map ({ha: 1, HA: 2}, ['HA', 'ha']); //=> [2, 1]
var mapAddTwo = map (function (x){ return x + 2});
//=> a function
```

filter

Callable -> List -> List
Callable -> Object -> Object
Callable -> String -> String

Filters a collection, removing all items that do not pass the test of applying the supplied callable to each item.

LiveScript

```
filter (< 3), [1 to 5] #=> [1, 2]
filter even, {a: 3, b: 4, c: 0} #=> {b: 4, c: 0}
filter (== \a), 'hahaha' #=> 'aaa'
```

JavaScript

```
filter(odd, [1, 2, 3, 4, 5]); //=> [1, 3, 5]
filter(function (x){ return x <= 2; }, {x: 4, y: 2, z: 8}); //=> {y: 2}
```

reject

Callable -> List -> List
Callable -> Object -> Object
Callable -> String -> String

Like `filter`, but instead of adding item to results it passes the test, adds those items who don't pass the test.

LiveScript

```
reject odd, [1 to 5] #=> [2, 4]
reject (-> typeof! it is \String), {a:
2, b: \ha}
#=> {a: 2}
reject (== \h), 'hmh     #=> 'm
```

JavaScript

```
reject(even, {a: 3, b: 8, c: 1}); //=>
{a: 3, c: 1}
```

partition

Callable -> List -> [List]
Callable -> Object -> [Object]
Callable -> String -> [String]

Equivalent to `[(filter f, xs), (reject f, xs)]`, but more efficient, only using one loop.

LiveScript

```
partition (in [\a to \c]), 'abcdefcf'
#=> ['abcc', 'deff']
```

JavaScript

```
partition(even, [1, 2, 3, 4, 5]);
//=> [[2, 4], [1, 3, 5]]
```

find

Callable -> List -> x
Callable -> Object -> x
Callable -> String -> x

Returns the first item to pass the test.

LiveScript

```
find odd, [2 4 6 7 8 9 10] #=> 7
```

JavaScript

```
find(function (x){ return  x <= 3; }, {b:
 3, a: 2});
//=> 2
```

each

Function -> List -> List
Function -> Object -> Object
Function -> String -> String

Applies function to each item in the collection (item in list or object, character in string), returns the original collection. Used for side effects.

LiveScript

```
each (.ps   \boom), [[\a] [\b] [\c]]
#=> [['a', 'boom'], ['b', 'boom'], ['c'
, 'boom']]
```

JavaScript

```
var count  = 4;
each(function (x){ count  += x; }, {a:1,
b:2, c:3});
count ; //=>  10
```

head

alias: first

`List -> x`
`String -> String`

The first item of the list, or first character of the string.

LiveScript	JavaScript
`head [1 to 5] #=> 1` `head 'hello'  #=> 'h'`	`head([9, 8, 7]); //=> 9` `head('world');  //-> 'w'`

tail

`List -> List`
`String -> String`

Everything but the first item of the list, or everything but the first character in the string.

LiveScript	JavaScript
`tail [1 to 5] #=> [2, 3, 4, 5]` `tail 'hello'  #=> 'ello'`	`tail([9, 8, 7]); //=> [8, 7]` `tail('world');  //-> 'orld'`

last

`List -> x`
`String -> String`

The last item of the list, or the last character of the string.

LiveScript	JavaScript
`last [1 to 5] #=> 5` `last 'hello'  #=> 'o'`	`last ([9, 8, 7]); //=> 7` `last ('world');  //-> 'd'`

initial

`List -> List`
`String -> String`

Everything but the last item of the list, or everything but the last character in the string.

LiveScript	JavaScript
`initial [1 to 5] #=> [1, 2, 3, 4]` `initial 'hello'  #=> 'hell'`	`initial([9, 8, 7]); //=> [9, 8]` `initial('world');  //-> 'worl'`

empty

`List -> Boolean`
`Object -> Boolean`

String -> Boolean

Whether the collection is empty, eg. `[]`, `{}`, or `''`.

LiveScript	JavaScript

`empty  [] #=> true`
`empty  {} #=> true`
`empty  '' #=> true`
	`empty ([]); //=> true`
	`empty ({}); //=> true`
	`empty (''); //=> true`

values

Object -> List

Returns a list of the values of the object.

LiveScript	JavaScript
`values  a: 2, b: 3, c: 9 #=> [2, 3, 9]`	`values ({mo : 'haha'}); //=> ['haha']`

keys

Object -> List

Returns a list of the keys of the object.

LiveScript	JavaScript
`keys a: 2, b: 3, c: 9 #=> ['a', 'b', 'c']`	`keys([1, 2, 3]); //=> [0, 1, 2]`

length

List -> Number
Object -> Number
String -> Number

Returns the number of items in the collection.

LiveScript	JavaScript

`length [1 4 2]     #=> 3`
`length h: 2, j: 23 #=> 2`
`length 'antwoord'  #=> 8`
	`length([3]); //=> 1`
	`length {}    //=> 0`
	`length 'doh' //=> 3`

cons

x -> List -> List
x -> y -> List
String -> String -> String

Returns a new list made by adding the supplied item to the front of the given list. Also mashes together strings.

LiveScript

```
cons 1 [2 3]  #=> [1 2 3]
cons 4 5      #=> [4 5]
cons 'a' 'bc' #=> 'abc'
```

JavaScript

```
cons(2, [4, 6]); //=> [2, 4, 6]
```

append

List -> List -> List
List -> x -> List
String -> String -> String

Returns a new list - either putting two lists together, or appending an item to a list. Also mashes together strings.

LiveScript

```
append [1 2] [3 4] #=> [1 2 3 4]
append [1 2] 3     #=> [1 2 3]
append 'abc' 'def' #=> 'abcdef'
```

JavaScript

```
append([4, 5], [6, 7]); //=> [4, 5, 6, 7]
append([4, 5], 6);      //=> [4, 5, 6]
append('ha', 'ha');     //=> 'haha'
```

join

String -> List -> String

Joins a list with the specified separator.

LiveScript

```
join \| [1 til 4] #=> '1|2|3'
```

JavaScript

```
join('*', [4, 5, 6]); //=> '4*5*6'
```

reverse

List -> List
String -> String

Reverses a list or a string.

LiveScript

```
reverse  [1 to 3] #=> [3, 2, 1]
reverse  'goat'   #=> 'taog'
```

JavaScript

```
reverse ([3, 4, 5]); //=> [5, 4, 3]
reverse ('Moo');     //=> 'ooM'
```

fold

alias: foldl

Function -> memo -> List -> x
Function -> memo -> Object -> x
Function -> memo -> String -> x

Takes a collection of items, and using the function supplied, folds them into a single value. Requires an initial value (memo), which will be the

result in case of an empty collection. The function supplied must take two arguments.

LiveScript

```
fold (+), 0, [1 to 5] #=> 15
product  = fold (*), 1
fold sumFunc , anObject
```

JavaScript

```
fold(function (x, y){ return  x + y; }, 0
,
  {a: 4, b: 5, c: 6}); //=> 15
```

fold1

alias: foldl1

Function -> List -> x
Function -> Object -> x
Function -> String -> x

Like `fold`, except assumes a non-empty collection, and thus doesn't require an initial value.

LiveScript

```
fold1 (+), [1 to 3] #=> 6
```

JavaScript

```
fold1(aFunction , aList );
```

foldr

Function -> memo -> List -> x
Function -> memo -> Object -> x
Function -> memo -> String -> x

Like `fold`, except from the right.

LiveScript

```
foldr (-), 9, [1 2 3 4] #=> -1
```

JavaScript

```
foldr(aFunction , initValue , aList );
```

foldr1

Function -> List -> x
Function -> Object -> x
Function -> String -> x

Like `foldr`, except assumes a non-empty collection, and thus doesn't require an initial value.

LiveScript

```
foldr1 (-), [1 2 3 4 9] #=> -1
```

JavaScript

```
foldr1(aFunction , aList );
```

unfold

alias: unfoldr

Function -> x -> List

Unfold builds a list from a seed value (`x`). It takes a function which either returns `null` if it is done producing the list, or returns `[a, b]`, `a` which is prepended to the list, and `b` is used as the next element in the recursive call.

LiveScript	JavaScript
`unfold (-> if it == 0 then null else [it, it - 1]), 10` `#=> [10,9,8,7,6,5,4,3,2,1]`	

andList

List -> Boolean
Object -> Boolean

Returns true if each item in the collection is true.

LiveScript	JavaScript
`andList [true, 2 + 2 == 4] #=> true` `andList [true, true, false] #=> false`	`andList (aListAllTrue); //=> true`

orList

List -> Boolean
Object -> Boolean

Returns true if any item in the collection is true.

LiveScript	JavaScript
`orList [false, false, true, false] #=> true`	`orList ({a: false, b: true}); //=> true`

any

Callable -> List -> Boolean
Callable -> Object -> Boolean
Callable -> String -> Boolean

Returns true if any of the items in the collection are true when applied to the test.

LiveScript	JavaScript
`any even, [3, 5, 7, 8, 9] #=> true`	`any(odd, [2, 4, 6, 8]); #=> false`

all

Callable -> List -> Boolean
Callable -> Object -> Boolean
Callable -> String -> Boolean

Returns true if all the items in the collection are true when applied to the test.

LiveScript

```
all (-> typeof! it is \String), <[ ha m a la ]>
#=> tre
```

JavaScript

```
all(fnction (x){ retrn  x == 'm ; }, 'm mm    );
//=> fals
```

unique

List -> List
Object -> List
String -> String

LiveScript

```
niqe   [1 1 1 3 3 6 7 8] #=> [1 3 6 7 8 ]
niqe   'aaabbccccd'      #=> 'abcd'
```

JavaScript

```
niqe  ([4, 4, 4, 4, 5]); //=> [4, 5]
```

sort

List -> List

Sorts a list, does not modify the input.

LiveScript

```
srt  [3 1 5 2 4 6]
#=> [1,2,3,4,5,6]
```

JavaScript

sortBy

(x -> y -> (-1 | 0 | 1)) -> List -> List

Takes a function which compares two items and returns either `-1`, `0`, or `1`, and a list, and sorts using that function. The original list is not modified.

LiveScript

```
obj = one: 1, two: 2, three: 3
srtBy  (cmpare  (obj.)), <[ three one two ]>
#=> ['one', 'two', 'three']
```

JavaScript

compare

Function -> x -> y -> (-1 | 0 | 1)

Takes a function, applies it to both `x` and `y`, and produces either `-1`, `0`, or `1`. Useful when combined with `sortBy`, and just given the first argument.

LiveScript

```
compare (.length), [1 to 3], [0 to 5] #
=> -1
compare (.length), [1 to 9], [0 to 5] #
=>  0
compare (.length), [1 to 4], [4 to 7] #
=>  1
```

JavaScript

sum

[Number] -> Number
{Number} -> Number

Sums up the values in the list or object. 0 if empty.

LiveScript

```
sum [1 to 5]          #=> 15
sum x: 1, y: 2, z: 3 #=> 6
```

JavaScript

```
sum ([4, 5, 6]); //=> 15
```

product

[Number] -> Number
{Number} -> Number

Gets the product of all the items in the list or objects. 1 if empty.

LiveScript

```
product [1 2 3] #=> 6
```

JavaScript

```
product ([4, 3, 2]); //=> 24
```

mean

alias: average

[Number] -> Number
{Number} -> Number

Gets the mean of the values in the list or object.

LiveScript

```
mean [1 to 5] #=> 3
```

JavaScript

```
mean ([4, 3]); //=> 3.5
```

concat

[List] -> List

Concatenates a list of lists together.

LiveScript

```
concat [[1] [2 3] [4]]   #=> [1, 2, 3, 4]
concat <[ aa bb cc dd ]> #=> 'aabbccdd'
```

JavaScript

```
concat([['a', 'b'], ['c']]); //=> ['a', 'b', 'c']
```

concatMap

Callable -> [List] -> List

Maps the callable on the list, then concats.

LiveScript

```
concatMap (-> [1 to it]), [1 2 3] #=> [1,1,2,1,2,3]
concatMap (.toUpperCase!),  <[ aa bb cc dd ]>
#=> 'AABBCCDD'
```

JavaScript

```
concatMap(function (x){ return  [x];}, [1, 2, 3]]);
//=> [1, 2, 3]
```

maximum

[Comparable] -> Comparable

Takes a list of comparable items, and returns the biggest of them.

LiveScript

```
maximum    [4 1 9 3] #=> 9
```

JavaScript

```
maximum   (['b', 'a', 'd']); //=> 'd'
```

minimum

[Comparable] -> Comparable

Takes a list of comparable items, and returns the smallest of them.

LiveScript

```
minimum    [\a to \f] #=> 'a'
```

JavaScript

```
minimum   ([3, 2, 8]); //=> 2
```

scan

alias: scanl

Function -> memo -> List -> List
Function -> memo -> Object -> List

Like a fold, except instead of just returning the final value, returns a list composed of the initial value, the intermediate values, and then the final value. Requires an initial value (memo) in case of an empty collection. The function supplied needs to take two arguments.

LiveScript	JavaScript
`scan (+), 0, [1 to 3] #=> [0, 1, 3, 6]`	`scan (someFunc , 0, [1, 2, 3]);`

scan1

alias: scanl1

Function -> List -> List
Function -> Object -> List

Like `scan`, except assumes non-empty collection, and thus doesn't require an initial value.

LiveScript	JavaScript
`scan1 (+), [1 to 3] #=> [1, 3, 6]`	`scan1 (someFunc , [1, 2, 3]);`

scanr

Function -> memo -> List -> List
Function -> memo -> Object -> List

Like `scan`, except from the right.

LiveScript	JavaScript
`scanr (+), 0, [1 to 3] #=> [6, 5, 3, 0]`	`scanr (someFunc , 0, [1, 2, 3]);`

scanr1

Function -> List -> List
Function -> Object -> List

Like `scanr`, except assumes non-empty collection, and thus doesn't require an initial value.

LiveScript	JavaScript
`scanr1 (+), [1 to 3] #=> [6, 5, 3]`	`scanr1 (someFunc , [1, 2, 3]);`

replicate

Number -> x -> List
Number -> String -> [String]

Takes its second argument, and repeats it n times.

LiveScript	JavaScript
`replicate 4 \a #=> ['a', 'a', 'a', 'a']`	`replicate(4, 2); //=> [3, 3, 3, 3]`

take

Number -> List -> List

Number -> String -> String

Returns the first n items in the list or string.

LiveScript

```
take 2 [1 to 5] #=> [1, 2]
take 4 \hello   #=> 'hell'
```

JavaScript

```
take(4, []); //=> []
var takeTwo = take(2);
```

drop

Number -> List -> List
Number -> String -> String

Returns the result of dropping the first n items of the list or string.

LiveScript

```
drop 2 [1 to 5] #=> [3, 4, 5]
drop 1 'goat'   #=> 'oat'
```

JavaScript

```
drop(6, [3, 4]); //=> []
drop(3, 'foot'); //=> 't'
```

splitAt

Number -> List -> [List]
Number -> String -> [String]

Equivalent to `[(take n, xs), (drop n, xs)]`

LiveScript

```
plitAt  2 [1 to 5] #=> [[1, 2], [3, 4, 5]]
```

JavaScript

```
plitAt (4, 'hello'); //=> ['hell', 'o']
```

takeWhile

Callable -> List -> List
Callable -> String -> String

Takes the first items of the list or string which pass the test.

LiveScript

```
takeWhile (in [\a to \d]), 'cabdek' #=>
 'cabd'
```

JavaScript

```
takeWhile(even, [2, 4, 5, 6]); //=> [2,
 4]
```

dropWhile

Callable -> List -> List
Callable -> String -> String

Drops the first items of the list or string which pass the test.

LiveScript

```
dropWhile even, [2 4 5 6] #=> [5, 6]
```

JavaScript

```
dropWhile(odd, [3, 1, 2, 5]); //=> [2, 5]
```

span

Callable -> List -> [List]
Callable -> String -> [String]

Equivalent to `[(takeWhile f, xs), (dropWhile f, xs)]`

LiveScript

```
span  even, [2 4 5 6] #=> [[2, 4], [5, 6]]
```

JavaScript

```
span (odd, [3, 1, 2, 5]); //=> [[3, 1], [2, 5]]
```

breakIt

Callable -> List -> [List]
Callable -> String -> [String]

Equivalent to `span (not) << f, xs`

LiveScript

```
breakIt (== 3), [1 to 5] #=> [[1, 2], [3, 4, 5]]
```

JavaScript

```
breakIt(even, [2, 4, 5, 6] //=> [[], [2, 4, 5, 6]]
```

listToObj

[List] -> Object

Converts a list of lists (pairs) into an object.

LiveScript

```
listToObj  [[\a 1] [\b 2]] #=> {a: 1, b: 2}
```

JavaScript

```
listToObj ([['d', 5], ['e', 6]]); //=> {d: 5, e: 6}
```

objToFunc

Object -> Function

Turns an object (anything with properties) into a function. This is used internally to allow objects to be accepted as callables.

LiveScript

```
f = objToFunc  m  4, c: 6
f \m #=> 4
```

JavaScript

```
var g = objToFunc (['a', 'b']);
g(1); //=> 'b'
```

zip

List -> List -> [List]

Zips together its arguments into a list of lists.

LiveScript

```
zip [1 2] [4 5]
#=> [[1, 4], [2, 5]]
```

JavaScript

```
zip([1, 2], [9, 8]);
//=> [[1, 9], [2, 8]]
```

zipWith

`Function -> List -> List -> List`

As `zip`, but applies the supplied function to the lists and creates a list of the results. The supplied function must take two arguments.

LiveScript

```
zipWith (+), [1 2 3], [3 2 1] #=> [4, 4
, 4]
zipWith (==), [4 5 3], [6 5 4]
#=> [false, true, false]
```

JavaScript

```
zipWith(binaryFunction , [3, 2, 1], [8,
2, 9]);
```

zipAll

`...Lists -> [List]`

Zips together its arguments into a list of lists. Can take any number of arguments, but is not curried.

LiveScript

```
zipAll [1 2 3] [4 5 6] [7 8 9]
#=> [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

JavaScript

```
zipAll([1, 2], [9, 8]);
//=> [[1, 9], [2, 8]]
```

zipAllWith

`Function -> ...Lists -> List`

As `zipAll`, but applies the supplied function to the lists and creates a list of the results. The supplied function must take in as many arguments as there are lists being inputed.

LiveScript

```
zipAllWith (-> &0 + &1 + &2), [1 2 3], [
3 2 1] [1 1 1]
#=> [5, 5, 5]
```

JavaScript

```
zipWith(binaryFunction , [3, 2, 1], [8,
2, 9]);
```

compose

`...Functions -> Function`

Compose takes a series of functions as arguments, and composes them, returning a single function which when called will apply all the functions. This is an alternative to doing `f1 >> f2 >> f3 >> f4` in LiveScript. Note that compose composes function forwards. The first argument will be the function called first, then the second, etc.

LiveScript

```
f = compose (+ 2), (* 2), (- 2)
f 3 #=> 8
```

JavaScript

```
var g = compose (f1, f2, f3, f4);
g();
```

curry

Function -> Function

Returns the function supplied, curried. Meaning, if the returned function is called with less than its number of arguments, it will return a new, partially applied function. This will only work on functions with a fixed length. All the functions in prelude.ls that can be curried are curried. You can define curried functions in LiveScript natively, but this function is useful if you have to deal with outside functions from libraries or such which you haven't defined.

LiveScript

```
curriedPow = curry Math.pow
fourToThe = curriedPow 4
fourToThe 2 #=> 16
```

JavaScript

```
var add = curry (function (x, y){ return
x + y; });
var addFour = add(4);
addFour (3); //=> 6
```

id

x -> x

A function which does nothing. It simply returns its argument.

LiveScript

```
id 5 #=> 5
```

JavaScript

```
id('something' ); //=> 'something'
```

flip

Function -> Function

Returns a function with the arguments flipped. Works with binary functions.

LiveScript

```
invertedPower = flip (^)
invertedPower 2 3 #=> 9
```

JavaScript

```
var divide = function (x, y){ return  x /
 y; };
var divideBy = flip(divide);
divideBy(2, 1); //=> 0.5
```

fix

(Function -> Function) -> Function

Fix-point function for anonymous recursion, implemented with the Y combinator.

LiveScript

```
(fix (fib) -> (n) ->
   | n <= 1      => 1
   | otherwise   => fib(n-1) + fib(n-2)
)(9) #=> 55
```

JavaScript

```
fix(function (fib){
    return  function (n){
      if (n <= 1) { return  1; }
      return  fib(n - 1) + fib(n - 2);
    };
  })(9); //=> 55
```

lines

String -> [String]

Splits a string at newlines into a list.

LiveScript

```
lines '''one
         two
         three'''
#=> ['one', 'two', 'three']
```

JavaScript

```
lines('a\nb\nc'); //=> ['a', 'b', 'c']
```

unlines

[String] -> String

Joins a list of strings into a single string using newlines.

LiveScript

```
unlines  [\one \two \three]
#=> 'one
#    two
#    three'
```

JavaScript

```
unlines (['a', 'b', 'c']); #=> 'a\nb\nc'
```

words

String -> [String]

Splits a string at spaces, returning a list of strings.

LiveScript

```
words 'hello, what isthat?'
#=> ['hello,', 'what', 'is', 'that?']
```

JavaScript

```
words('at the end  of   the');
//=> ['at', 'the', 'end', 'of', 'the']
```

unwords

`[String] -> String`

Joins a list of strings into a single string using spaces.

LiveScript

```
unwords  [\one \two \three] #=> 'one two
 three'
```

JavaScript

```
unwords (['a', 'b', 'c']); //=> 'a b c'
```

max

`Comparable -> Comparable -> Value`

Takes two arguments which can be compared using `>`, returns the larger one.

LiveScript

```
max  3 1   #=> 3
max  \a \c #=> "c"
```

JavaScript

```
max (4, 5);     //=> 5
max ('d', 'g'); //=> "g"
```

min

`Comparable -> Comparable -> Value`

Takes two arguments which can be compared using `>`, returns the smaller one.

LiveScript

```
min  3 1   #=> 1
min  \a \c #=> "a"
```

JavaScript

```
min (4, 5);     //=> 4
min ('d', 'g'); //=> "d"
```

negate

`Negatable -> Number`

Takes an argument, most often a number, but also anything that can be coerced into a number, eg. `'3'`, and returns it as a number, negated.

LiveScript

```
negate 3  #=> -3
negate -2 #=>  2
```

JavaScript

```
negate(-1);  //=>  1
negate('5'); //=> -5
```

abs

`Number -> Number`

Takes a number and returns its absolute value. A number's absolute value is

LiveScript	JavaScript
`abs -2 #=>  2` `abs 2  #=> 2`	`negate(-1);  //=>  1` `negate('5'); //=> -5`

signum

`Number -> (-1 | 0 | 1)`

Takes a number and returns either -1, 0, or 1 depending on the sign of the number.

LiveScript	JavaScript

`signum  -5 #=> -1`
`signum   0 #=>  0`
`signum   9 #=>  1`
	`signum  (-1); //=> -1`
	`signum  ( 0); //=>  0`
	`signum  ( 1); //=>  1`

quot

`Number -> Number -> Number`

Division truncated toward 0.

LiveScript	JavaScript
`quot  -20 3 #=> -6`	`quot (-20, 3); //=> -6`

rem

`Number -> Number -> Number`

Remainder, like the `%` operator.

LiveScript	JavaScript
`rem -20 3 #=> -2`	`rem(-20, 3); //=> -2`

div

`Number -> Number -> Number`

Division truncated down toward negative infinity.

LiveScript	JavaScript
`div -20 3 #=> -7`	`div(-20, 3); //=> -7`

mod

`Number -> Number -> Number`

Remainder, like the `%%` operator in LiveScript.

LiveScript	JavaScript
`md  -20 3 #=> 1`	`md (-20, 3); //=> -1`

recip

`Number -> Number`

One over the number, ie `1 / x`

LiveScript	JavaScript
`recip 4 #=> 0.25`	`recip(0.5); //=> 2`

pi

`Number`

π

LiveScript	JavaScript
`pi #=> 3.141592653589793`	`pi; //=> 3.141592653589793`

tau

`Number`

τ - equal to 2 * π

LiveScript	JavaScript
`tau #=> 6.283185307179586`	`tau; //=> 6.283185307179586`

exp

`Number -> Number`

Returns e to the argument.

LiveScript	JavaScript
`exp 1 #=> 2.718281828459045`	`exp(1); #=> 2.718281828459045`

sqrt

`Number -> Number`

Square root.

LiveScript	JavaScript
`sqrt  4 #=> 2`	`sqrt (16); //=> 4`

ln

Natural log.

`Number -> Number`

LiveScript	JavaScript
`ln 10 #=> 2.302585092994046`	`ln(10); //=> 2.302585092994046`

pow

`Number -> Number -> Number`

Power. Equivalent to `x ^ y` or `x ** y` in LiveScript.

LiveScript	JavaScript
`pow -2 2 #=> 4`	`pow(4, 2) #=> 16`

sin

`Number -> Number`

cos

`Number -> Number`

tan

`Number -> Number`

asin

`Number -> Number`

acos

`Number -> Number`

atan

`Number -> Number`

atan2

`Number -> Number -> Number`

truncate

`Number -> Number`

Number truncated toward 0.

LiveScript	JavaScript
`truncate -1.5 #=> -1` `truncate  1.5 #=>  1`	`truncate (-1.8); //=> -1` `truncate ( 1.2); //=>  1`

round

`Number -> Number`

Number rounded to nearest whole number.

LiveScript	JavaScript

`round  0.6 #=> 1`
`round  0.5 #=> 1`
`round  0.4 #=> 0`
	`round (-1.3); //=> -1`
	`round (-1.8); //=> -2`
	`round ( 9.5); //=>  10`

ceiling

`Number -> Number`

Number rounded up.

LiveScript	JavaScript
`ceiling 0.1 #=> 1`	`ceiling(-0.9); //=> 0`

floor

`Number -> Number`

Number rounded down.

LiveScript	JavaScript
`floor 0.9 #=> 0`	`floor(-0.1); //=> -1`

isItNaN

`Number -> Boolean`

Is it `NaN` (not a number)? More accurate than the native `isNaN` function.

LiveScript	JavaScript
`isItNaN  sqrt  -1 #=> true`	`isItNaN (undefined ); //=> false`

even

`Number -> Boolean`

Is the number even?

LiveScript	JavaScript
`even 4 #=> tru`	`even(0); //=> tru`

odd

`Number -> Boolean`

Is the number odd?

LiveScript	JavaScript
`odd 3 #=> tru`	`odd(0); //=> fals`

gcd

`Number -> Number -> Number`

Greatest common denominator.

LiveScript	JavaScript
`gcd 12 18 #=> 6`	`gcd(12, 18); //=> 6`

lcm

`Number -> Number -> Number`

Least common multiple.

LiveScript	JavaScript
`lcm 12 18 #=> 36`	`lcm(12, 18); //=> 36`