LiveScript is a functional compile-to-JavaScript language which shares most of the underlying semantics with its host language. Nice additions comes with currying, function composition, pattern matching and lots of other goodies heavily borrowed from languages like Haskell, F# and Scala.

LiveScript is a fork of Coco, which is itself a fork of CoffeeScript. The language is stable, and a new version is in active development to bring a plethora of new niceties!

Feedback is always welcome, so feel free to reach me over at [@kurisuwhyte](https://twitter.com/kurisuwhyte) :)

```
# Just like its CoffeeScript cousin, LiveScript uses number symbols for
# single-line comments.

/*
 Multi-line comments are written C-style. Use them if you want comments
 to be preserved in the JavaScript output.
 */

# As far as syntax goes, LiveScript uses indentation to delimit blocks,
# rather than curly braces, and whitespace to apply functions, rather
# than parenthesis.


########################################################################
## 1. Basic values
########################################################################

# Lack of value is defined by the keyword `void` instead of `undefined`
void            # same as `undefined` but safer (can't be overridden)

# No valid value is represented by Null.
null


# The most basic actual value is the logical type:
true
false

# And it has a plethora of aliases that mean the same thing:
on; off
yes; no


# Then you get numbers. These are double-precision floats like in JS.
10
0.4     # Note that the leading `0` is required

# For readability, you may use underscores and letter suffixes in a
# number, and these will be ignored by the compiler.
12_344km


# Strings are immutable sequences of characters, like in JS:
"Christina"             # apostrophes are okay too!
"""Multi-line
```

```
   strings
   are
   okay
   too."""

# Sometimes you want to encode a keyword, the backslash notation makes
# this easy:
\keyword                # => 'keyword'


# Arrays are ordered collections of values.
fruits =
  * \apple
  * \orange
  * \pear

# They can be expressed more concisely with square brackets:
fruits = [ \apple, \orange, \pear ]

# You also get a convenient way to create a list of strings, using
# white space to delimit the items.
fruits = <[ apple orange pear ]>

# You can retrieve an item by their 0-based index:
fruits[0]       # => "apple"

# Objects are a collection of unordered key/value pairs, and a few other
# things (more on that later).
person =
  name: "Christina"
  likes:
    * "kittens"
    * "and other cute stuff"

# Again, you can express them concisely with curly brackets:
person = {name: "Christina", likes: ["kittens", "and other cute stuff"]}

# You can retrieve an item by their key:
person.name     # => "Christina"
person["name"]  # => "Christina"


# Regular expressions use the same syntax as JavaScript:
trailing-space = /\s$/          # dashed-words become dashedWords

# Except you can do multi-line expressions too!
# (comments and whitespace just gets ignored)
funRE = //
        function\s+(.+)         # name
        \s* \((.*)\) \s*        # arguments
        { (.*) }                # body
        //
```

```
########################################################################
## 2. Basic operations
########################################################################

# Arithmetic operators are the same as JavaScript's:
1 + 2   # => 3
2 - 1   # => 1
2 * 3   # => 6
4 / 2   # => 2
3 % 2   # => 1


# Comparisons are mostly the same too, except that `==` is the same as
# JS's `===`, where JS's `==` in LiveScript is `~=`, and `===` enables
# object and array comparisons, and also stricter comparisons:
2 == 2          # => true
2 == "2"        # => false
2 ~= "2"        # => true
2 === "2"       # => false

[1,2,3] == [1,2,3]        # => false
[1,2,3] === [1,2,3]       # => true

+0 == -0     # => true
+0 === -0    # => false

# Other relational operators include <, <=, > and >=

# Logical values can be combined through the logical operators `or`,
# `and` and `not`
true and false  # => false
false or true   # => true
not false       # => true


# Collections also get some nice additional operators
[1, 2] ++ [3, 4]                # => [1, 2, 3, 4]
'a' in <[ a b c ]>              # => true
'name' of { name: 'Chris' }     # => true


########################################################################
## 3. Functions
########################################################################

# Since LiveScript is functional, you'd expect functions to get a nice
# treatment. In LiveScript it's even more apparent that functions are
# first class:
add = (left, right) -> left + right
add 1, 2        # => 3

# Functions which take no arguments are called with a bang!
two = -> 2
two!
```

```
# LiveScript uses function scope, just like JavaScript, and has proper
# closures too. Unlike JavaScript, the `=` works as a declaration
# operator, and will always declare the variable on the left hand side.

# The `:=` operator is available to *reuse* a name from the parent
# scope.


# You can destructure arguments of a function to quickly get to
# interesting values inside a complex data structure:
tail = ([head, ...rest]) -> rest
tail [1, 2, 3]  # => [2, 3]

# You can also transform the arguments using binary or unary
# operators. Default arguments are also possible.
foo = (a = 1, b = 2) -> a + b
foo!    # => 3

# You could use it to clone a particular argument to avoid side-effects,
# for example:
copy = (^^target, source) ->
  for k,v of source => target[k] = v
  target
a = { a: 1 }
copy a, { b: 2 }        # => { a: 1, b: 2 }
a                       # => { a: 1 }


# A function may be curried by using a long arrow rather than a short
# one:
add = (left, right) --> left + right
add1 = add 1
add1 2          # => 3

# Functions get an implicit `it` argument, even if you don't declare
# any.
identity = -> it
identity 1      # => 1

# Operators are not functions in LiveScript, but you can easily turn
# them into one! Enter the operator sectioning:
divide-by-two = (/ 2)
[2, 4, 8, 16].map(divide-by-two) .reduce (+)


# Not only of function application lives LiveScript, as in any good
# functional language you get facilities for composing them:
double-minus-one = (- 1) . (* 2)

# Other than the usual `f . g` mathematical formulae, you get the `>>`
# and `<<` operators, that describe how the flow of values through the
# functions.
double-minus-one = (* 2) >> (- 1)
```

```
double-minus-one = (- 1) << (* 2)


# And talking about flow of value, LiveScript gets the `|>` and `<|`
# operators that apply a value to a function:
map = (f, xs) --> xs.map f
[1 2 3] |> map (* 2)            # => [2 4 6]

# You can also choose where you want the value to be placed, just mark
# the place with an underscore (_):
reduce = (f, xs, initial) --> xs.reduce f, initial
[1 2 3] |> reduce (+), _, 0     # => 6


# The underscore is also used in regular partial application, which you
# can use for any function:
div = (left, right) -> left / right
div-by-two = div _, 2
div-by-two 4      # => 2


# Last, but not least, LiveScript has back-calls, which might help
# with some callback-based code (though you should try more functional
# approaches, like Promises):
readFile = (name, f) -> f name
a <- readFile 'foo'
b <- readFile 'bar'
console.log a + b

# Same as:
readFile 'foo', (a) -> readFile 'bar', (b) -> console.log a + b


########################################################################
## 4. Patterns, guards and control-flow
########################################################################

# You can branch computations with the `if...else` expression:
x = if n > 0 then \positive else \negative

# Instead of `then`, you can use `=>`
x = if n > 0 => \positive
    else        \negative

# Complex conditions are better-off expressed with the `switch`
# expression, though:
y = {}
x = switch
  | (typeof y) is \number => \number
  | (typeof y) is \string => \string
  | 'length' of y         => \array
  | otherwise             => \object      # `otherwise` and `_` always matches.

# Function bodies, declarations and assignments get a free `switch`, so
# you don't need to type it again:
take = (n, [x, ...xs]) -->
                        | n == 0 => []
                        | _      => [x] ++ take (n - 1), xs


########################################################################
## 5. Comprehensions
########################################################################

# While the functional helpers for dealing with lists and objects are
# right there in the JavaScript's standard library (and complemented on
# the prelude-ls, which is a "standard library" for LiveScript),
# comprehensions will usually allow you to do this stuff faster and with
# a nice syntax:
oneToTwenty = [1 to 20]
evens       = [x for x in oneToTwenty when x % 2 == 0]

# `when` and `unless` can be used as filters in the comprehension.

# Object comprehension works in the same way, except that it gives you
# back an object rather than an Array:
copy = { [k, v] for k, v of source }


########################################################################
## 4. OOP
########################################################################

# While LiveScript is a functional language in most aspects, it also has
# some niceties for imperative and object oriented programming. One of
# them is class syntax and some class sugar inherited from CoffeeScript:
class Animal
  (@name, kind) ->
    @kind = kind
  action: (what) -> "*#{@name} (a #{@kind}) #{what}*"

class Cat extends Animal
  (@name) -> super @name, 'cat'
  purr: -> @action 'purrs'

kitten = new Cat 'Mei'
kitten.purr!      # => "*Mei (a cat) purrs*"

# Besides the classical single-inheritance pattern, you can also provide
# as many mixins as you would like for a class. Mixins are just plain
# objects:
Huggable =
  hug: -> @action 'is hugged'

class SnugglyCat extends Cat implements Huggable

kitten = new SnugglyCat 'Purr'
kitten.hug!     # => "*Mei (a cat) is hugged*"
```

Further reading

There's just so much more to LiveScript, but this should be enough to get you started writing little functional things in it. The official website has a lot of information on the language, and a nice online compiler for you to try stuff out!

You may also want to grab yourself some prelude.ls, and check out the `#livescript` channel on the Freenode network.