CoffeeScript is a little language that compiles into JavaScript. Underneath all of those embarrassing braces and semicolons, JavaScript has always had a gorgeous object model at its heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way.

The golden rule of CoffeeScript is: *"It's just JavaScript"*. The code compiles one-to-one into the equivalent JS, and there is no interpretation at runtime. You can use any existing JavaScript library seamlessly (and vice-versa). The compiled output is readable and pretty-printed, passes through JavaScript Lint without warnings, will work in every JavaScript implementation, and tends to run as fast or faster than the equivalent handwritten JavaScript.

Latest Version: 1.1.0

Overview

CoffeeScript on the left, compiled JavaScript output on the right.

```
# Assignment:
number   = 42
opposite = true

# Conditions:
number = -42 if opposite

# Functions:
square = (x) -> x * x

# Arrays:
list = [1, 2, 3, 4, 5]

# Objects:
math =
  root:   Math.sqrt
  square: square
  cube:   (x) -> x * square x

# Splats:
race = (winner, runners...) ->
  print winner, runners

# Existence:
alert "I knew it!" if elvis?

# Array comprehensions:
cubes = (math.cube num for num in list)
```

```
var cubes, list, math, num, number, opposite, race, square;
var __slice = Array.prototype.slice;
number = 42;
opposite = true;
if (opposite) {
  number = -42;
}
square = function(x) {
  return x * x;
};
list = [1, 2, 3, 4, 5];
math = {
  root: Math.sqrt,
  square: square,
  cube: function(x) {
    return x * square(x);
  }
};
race = function() {
  var runners, winner;
  winner = arguments[0], runners = 2 <= arguments.length ?
__slice.call(arguments, 1) : [];
  return print(winner, runners);
};
if (typeof elvis !== "undefined" && elvis !== null) {
  alert("I knew it!");
}
cubes = (function() {
  var _i, _len, _results;
  _results = [];
  for (_i = 0, _len = list.length; _i < _len; _i++) {
    num = list[_i];
    _results.push(math.cube(num));
  }
  return _results;
})();
```

run: cubes

Installation and Usage

The CoffeeScript compiler is itself written in CoffeeScript, using the Jison parser generator. The command-line version of `coffee` is available as a Node.js utility. The core compiler however, does not depend on Node, and can be run in any JavaScript environment, or in the browser (see "Try CoffeeScript", above).

To install, first make sure you have a working copy of the latest stable version of Node.js,

and npm (the Node Package Manager). You can then install CoffeeScript with npm:

```
npm install -g coffee-script
```

(Leave off the `-g` if you don't wish to install globally.)

If you'd prefer to install the latest master version of CoffeeScript, you can clone the CoffeeScript source repository from GitHub, or download the source directly. To install the CoffeeScript compiler system-wide under `/usr/local`, open the directory and run:

```
sudo bin/cake install
```

If installing on Ubuntu or Debian, be careful not to use the existing out-of-date package. If installing on Windows, your best bet is probably to run Node.js under Cygwin.

Once installed, you should have access to the `coffee` command, which can execute scripts, compile `.coffee` files into `.js`, and provide an interactive REPL. The `coffee` command takes the following options:

`-c, --compile`	Compile a `.coffee` script into a `.js` JavaScript file of the same name.
`-i, --interactive`	Launch an interactive CoffeeScript session to try short snippets. More pleasant if wrapped with rlwrap.
`-o, --output [DIR]`	Write out all compiled JavaScript files into the specified directory. Use in conjunction with `--compile` or `--watch`.
`-j, --join [FILE]`	Before compiling, concatenate all scripts together in the order they were passed, and write them into the specified file. Useful for building large projects.
`-w, --watch`	Watch the modification times of the coffee-scripts, recompiling as soon as a change occurs.
`-p, --print`	Instead of writing out the JavaScript as a file, print it directly to **stdout**.
`-l, --lint`	If the `jsl` (JavaScript Lint) command is installed, use it to check the compilation of a CoffeeScript file. (Handy in conjunction with `--watch`)
`-s, --stdio`	Pipe in CoffeeScript to STDIN and get back JavaScript over STDOUT. Good for use with processes written in other languages. An example: `cat src/cake.coffee | coffee -sc`
`-e, --eval`	Compile and print a little snippet of CoffeeScript directly from the command line. For example: `coffee -e "puts num for num in [10..1]"`
`-r, --require`	Load a library before compiling or executing your script. Can be used to hook in to the compiler (to add Growl notifications, for example).
`-b, --bare`	Compile the JavaScript without the top-level function safety wrapper. (Used for CoffeeScript as a Node.js module.)
`-t, --tokens`	Instead of parsing the CoffeeScript, just lex it, and print out the token stream: `[IDENTIFIER square] [ASSIGN =] [PARAM_START (]` ...
`-n, --nodes`	Instead of compiling the CoffeeScript, just lex and parse it, and print out the parse tree: `Expressions`

```
Assign
  Value "square"
  Code "x"
    Op *
      Value "x"
      Value "x"
```

`--nodejs`	The `node` executable has some useful options you can set, such as `--debug` and `--max-stack-size`. Use this flag to forward options directly to Node.js.

Examples:

- Compile a directory tree of `.coffee` files into a parallel tree of `.js`, in `lib`:
  `coffee -o lib/ -c src/`
- Watch a file for changes, and recompile it every time the file is saved:
  `coffee --watch --compile experimental.coffee`
- Concatenate a list of files into a single script:
  `coffee --join project.js --compile src/*.coffee`
- Print out the compiled JS from a one-liner:
  `coffee -bpe "alert i for i in [0..10]"`
- Start the CoffeeScript REPL:
  `coffee`

Language Reference

This reference is structured so that it can be read from top to bottom, if you like. Later sections use ideas and syntax previously introduced. Familiarity with JavaScript is assumed. In all of the following examples, the source CoffeeScript is provided on the left, and the direct compilation into JavaScript is on the right.

*Many of the examples can be run (where it makes sense) by pressing the **run** button on the right, and can be loaded into the "Try CoffeeScript" console by pressing the **load** button on the left.*

First, the basics: CoffeeScript uses significant whitespace to delimit blocks of code. You don't need to use semicolons `;` to terminate expressions, ending the line will do just as well, (although semicolons can still be used to fit multiple expressions onto a single line.) Instead of using curly braces `{ }` to surround blocks of code in functions, if-statements, switch, and try/catch, use indentation.

You don't need to use parentheses to invoke a function if you're passing arguments. The implicit call wraps forward to the end of the line or block expression.
`console.log sys.inspect object` → `console.log(sys.inspect(object));`

Functions

Functions are defined by an optional list of parameters in parentheses, an arrow, and the function body. The empty function looks like this: `->`

```
square = (x) -> x * x
cube   = (x) -> square(x) * x
```

load

```
var cube, square;
square = function(x) {
  return x * x;
};
cube = function(x) {
  return square(x) * x;
};
```

run: cube(5)

Functions may also have default values for arguments. Override the default value by passing a non-null argument.

```coffeescript
fill = (container, liquid = "coffee") ->
  "Filling the #{container} with #{liquid}..."
```

```javascript
var fill;
fill = function(container, liquid) {
  if (liquid == null) {
    liquid = "coffee";
  }
  return "Filling the " + container + " with " + liquid + "...";
};
```

load run: fill("cup")

Objects and Arrays

The CoffeeScript literals for objects and arrays look very similar to their JavaScript cousins. When each property is listed on its own line, the commas are optional. Objects may be created using indentation instead of explicit braces, similar to YAML.

```coffeescript
song = ["do", "re", "mi", "fa", "so"]

singers = {Jagger: "Rock", Elvis: "Roll"}

bitlist = [
  1, 0, 1
  0, 0, 1
  1, 1, 0
]

kids =
  brother:
    name: "Max"
    age:  11
  sister:
    name: "Ida"
    age:  9
```

```javascript
var bitlist, kids, singers, song;
song = ["do", "re", "mi", "fa", "so"];
singers = {
  Jagger: "Rock",
  Elvis: "Roll"
};
bitlist = [1, 0, 1, 0, 0, 1, 1, 1, 0];
kids = {
  brother: {
    name: "Max",
    age: 11
  },
  sister: {
    name: "Ida",
    age: 9
  }
};
```

load run: song.join(" ... ")

In JavaScript, you can't use reserved words, like `class`, as properties of an object, without quoting them as strings. CoffeeScript notices reserved words used as keys in objects and quotes them for you, so you don't have to worry about it (say, when using jQuery).

```coffeescript
$('.account').attr class: 'active'

log object.class
```

```javascript
$('.account').attr({
  "class": 'active'
});
log(object["class"]);
```

load

Lexical Scoping and Variable Safety

The CoffeeScript compiler takes care to make sure that all of your variables are properly declared within lexical scope — you never need to write `var` yourself.

```coffeescript
outer = 1
changeNumbers = ->
  inner = -1
  outer = 10
inner = changeNumbers()
```

```javascript
var changeNumbers, inner, outer;
outer = 1;
changeNumbers = function() {
  var inner;
  inner = -1;
  return outer = 10;
};
inner = changeNumbers();
```

load run: inner

Notice how all of the variable declarations have been pushed up to the top of the closest scope, the first time they appear. **outer** is not redeclared within the inner function, because it's already in scope; **inner** within the function, on the other hand, should not be able to change the value of the external variable of the same name, and therefore has a declaration of its own.

This behavior is effectively identical to Ruby's scope for local variables. Because you don't have direct access to the `var` keyword, it's impossible to shadow an outer variable on purpose, you may only refer to it. So be careful that you're not reusing the name of an external variable accidentally, if you're writing a deeply nested function.

Although suppressed within this documentation for clarity, all CoffeeScript output is wrapped in an anonymous function: `(function(){ ... })();` This safety wrapper, combined with the automatic generation of the `var` keyword, make it exceedingly difficult to pollute the global namespace by accident.

If you'd like to create top-level variables for other scripts to use, attach them as properties on **window**, or on the **exports** object in CommonJS. The **existential operator** (covered below), gives you a reliable way to figure out where to add them, if you're targeting both CommonJS and the browser: `exports ? this`

If, Else, Unless, and Conditional Assignment

If/else statements can be written without the use of parentheses and curly brackets. As with functions and other block expressions, multi-line conditionals are delimited by indentation. There's also a handy postfix form, with the `if` or `unless` at the end.

CoffeeScript can compile **if** statements into JavaScript expressions, using the ternary operator when possible, and closure wrapping otherwise. There is no explicit ternary statement in CoffeeScript — you simply use a regular **if** statement on a single line.

```
mood = greatlyImproved if singing

if happy and knowsIt
  clapsHands()
  chaChaCha()
else
  showIt()

date = if friday then sue else jill

options or= defaults
```

```
var date, mood;
if (singing) {
  mood = greatlyImproved;
}
if (happy && knowsIt) {
  clapsHands();
  chaChaCha();
} else {
  showIt();
}
date = friday ? sue : jill;
options || (options = defaults);
```

load

Splats...

The JavaScript **arguments object** is a useful way to work with functions that accept variable numbers of arguments. CoffeeScript provides splats `...`, both for function definition as well as invocation, making variable numbers of arguments a little bit more palatable.

```
gold = silver = rest = "unknown"

awardMedals = (first, second, others...) ->
  gold   = first
  silver = second
  rest   = others

contenders = [
  "Michael Phelps"
  "Liu Xiang"
  "Yao Ming"
  "Allyson Felix"
  "Shawn Johnson"
  "Roman Sebrle"
  "Guo Jingjing"
  "Tyson Gay"
  "Asafa Powell"
  "Usain Bolt"
]
```

```
var awardMedals, contenders, gold, rest, silver;
var __slice = Array.prototype.slice;
gold = silver = rest = "unknown";
awardMedals = function() {
  var first, others, second;
  first = arguments[0], second = arguments[1], others = 3 <=
arguments.length ? __slice.call(arguments, 2) : [];
  gold = first;
  silver = second;
  return rest = others;
};
contenders = ["Michael Phelps", "Liu Xiang", "Yao Ming",
"Allyson Felix", "Shawn Johnson", "Roman Sebrle", "Guo
Jingjing", "Tyson Gay", "Asafa Powell", "Usain Bolt"];
awardMedals.apply(null, contenders);
alert("Gold: " + gold);
alert("Silver: " + silver);
alert("The Field: " + rest);
```

```
awardMedals contenders...

alert "Gold: " + gold
alert "Silver: " + silver
alert "The Field: " + rest
```

load run

Loops and Comprehensions

Most of the loops you'll write in CoffeeScript will be **comprehensions** over arrays, objects, and ranges. Comprehensions replace (and compile into) **for** loops, with optional guard clauses and the value of the current array index. Unlike for loops, array comprehensions are expressions, and can be returned and assigned.

```
# Eat lunch.
eat food for food in ['toast', 'cheese', 'wine']
```

load

```
var food, _i, _len, _ref;
_ref = ['toast', 'cheese', 'wine'];
for (_i = 0, _len = _ref.length; _i < _len; _i++) {
  food = _ref[_i];
  eat(food);
}
```

Comprehensions should be able to handle most places where you otherwise would use a loop, **each**/**forEach**, **map**, or **select**/**filter**:
`shortNames = (name for name in list when name.length < 5)`
If you know the start and end of your loop, or would like to step through in fixed-size increments, you can use a range to specify the start and end of your comprehension.

```
countdown = (num for num in [10..1])
```

load

```
var countdown, num;
countdown = (function() {
  var _results;
  _results = [];
  for (num = 10; num >= 1; num--) {
    _results.push(num);
  }
  return _results;
})();
```

run: countdown

Note how because we are assigning the value of the comprehensions to a variable in the example above, CoffeeScript is collecting the result of each iteration into an array. Sometimes functions end with loops that are intended to run only for their side-effects. Be careful that you're not accidentally returning the results of the comprehension in these cases, by adding a meaningful return value, like `true`, or `null`, to the bottom of your function.

To step through a range comprehension in fixed-size chunks, use `by`, for example:
`evens = (x for x in [0..10] by 2)`

Comprehensions can also be used to iterate over the keys and values in an object. Use `of` to signal comprehension over the properties of an object instead of the values in an array.

```
yearsOld = max: 10, ida: 9, tim: 11

ages = for child, age of yearsOld
  child + " is " + age
```

load

```
var age, ages, child, yearsOld;
yearsOld = {
  max: 10,
  ida: 9,
  tim: 11
};
ages = (function() {
  var _results;
  _results = [];
  for (child in yearsOld) {
    age = yearsOld[child];
    _results.push(child + " is " + age);
  }
  return _results;
})();
```

run: ages.join(", ")

If you would like to iterate over just the keys that are defined on the object itself, by adding a `hasOwnProperty` check to avoid properties that may be interited from the prototype, use `for own key, value of object`

The only low-level loop that CoffeeScript provides is the **while** loop. The main difference from JavaScript is that the **while** loop can be used as an expression, returning an array containing the result of each iteration through the loop.

```
# Econ 101
if this.studyingEconomics
  buy()  while supply > demand
  sell() until supply > demand

# Nursery Rhyme
num = 6
lyrics = while num -= 1
  num + " little monkeys, jumping on the bed.
    One fell out and bumped his head."
```

load

```
var lyrics, num;
if (this.studyingEconomics) {
  while (supply > demand) {
    buy();
  }
  while (!(supply > demand)) {
    sell();
  }
}
num = 6;
lyrics = (function() {
  var _results;
  _results = [];
  while (num -= 1) {
    _results.push(num + " little monkeys, jumping on the bed.
One fell out and bumped his head.");
  }
  return _results;
})();
```

run: lyrics.join("\n")

For readability, the **until** keyword is equivalent to `while not`, and the **loop** keyword is equivalent to `while true`.

When using a JavaScript loop to generate functions, it's common to insert a closure wrapper in order to ensure that loop variables are closed over, and all the generated functions don't just share the final values. CoffeeScript provides the `do` keyword, which immediately invokes a passed function, forwarding any arguments.

```
for filename in list
  do (filename) ->
    fs.readFile filename, (err, contents) ->
      compile filename, contents.toString()
```

load

```
var filename, _fn, _i, _len;
_fn = function(filename) {
  return fs.readFile(filename, function(err, contents) {
    return compile(filename, contents.toString());
  });
};
for (_i = 0, _len = list.length; _i < _len; _i++) {
  filename = list[_i];
  _fn(filename);
}
```

Array Slicing and Splicing with Ranges

Ranges can also be used to extract slices of arrays. With two dots (`3..6`), the range is inclusive (`3, 4, 5, 6`); with three dots (`3...6`), the range excludes the end (`3, 4, 5`).

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

copy    = numbers[0...numbers.length]

middle  = copy[3..6]
```

load

```
var copy, middle, numbers;
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
copy = numbers.slice(0, numbers.length);
middle = copy.slice(3, 7);
```

run: middle

The same syntax can be used with assignment to replace a segment of an array with new values, splicing it.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
var numbers, _ref;
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

```
numbers[3..6] = [-3, -4, -5, -6]
```

load

```
[].splice.apply(numbers, [3, 4].concat(_ref = [-3, -4, -5,
-6])), _ref;
```

run: numbers

Note that JavaScript strings are immutable, and can't be spliced.

Everything is an Expression (at least, as much as possible)

You might have noticed how even though we don't add return statements to CoffeeScript functions, they nonetheless return their final value. The CoffeeScript compiler tries to make sure that all statements in the language can be used as expressions. Watch how the `return` gets pushed down into each possible branch of execution, in the function below.

```
grade = (student) ->
  if student.excellentWork
    "A+"
  else if student.okayStuff
    if student.triedHard then "B" else "B-"
  else
    "C"

eldest = if 24 > 21 then "Liz" else "Ike"
```

load

```
var eldest, grade;
grade = function(student) {
  if (student.excellentWork) {
    return "A+";
  } else if (student.okayStuff) {
    if (student.triedHard) {
      return "B";
    } else {
      return "B-";
    }
  } else {
    return "C";
  }
};
eldest = 24 > 21 ? "Liz" : "Ike";
```

run: eldest

Even though functions will always return their final value, it's both possible and encouraged to return early from a function body writing out the explicit return (`return value`), when you know that you're done.

Because variable declarations occur at the top of scope, assignment can be used within expressions, even for variables that haven't been seen before:

```
six = (one = 1) + (two = 2) + (three = 3)
```

load

```
var one, six, three, two;
six = (one = 1) + (two = 2) + (three = 3);
```

run: six

Things that would otherwise be statements in JavaScript, when used as part of an expression in CoffeeScript, are converted into expressions by wrapping them in a closure. This lets you do useful things, like assign the result of a comprehension to a variable:

```
# The first ten global properties.

globals = (name for name of window)[0...10]
```

load

```
var globals, name;
globals = ((function() {
  var _results;
  _results = [];
  for (name in window) {
    _results.push(name);
  }
  return _results;
})()).slice(0, 10);
```

run: globals

As well as silly things, like passing a **try/catch** statement directly into a function call:

```
alert(
  try
    nonexistent / undefined
  catch error
    "And the error is ... " + error
)
```

load

```
alert((function() {
  try {
    return nonexistent / void 0;
  } catch (error) {
    return "And the error is ... " + error;
  }
})());
```

run

There are a handful of statements in JavaScript that can't be meaningfully converted into

expressions, namely `break`, `continue`, and `return`. If you make use of them within a block of code, CoffeeScript won't try to perform the conversion.

Operators and Aliases

Because the `==` operator frequently causes undesirable coercion, is intransitive, and has a different meaning than in other languages, CoffeeScript compiles `==` into `===`, and `!=` into `!==`. In addition, `is` compiles into `===`, and `isnt` into `!==`.

You can use `not` as an alias for `!`.

For logic, `and` compiles to `&&`, and `or` into `||`.

Instead of a newline or semicolon, `then` can be used to separate conditions from expressions, in **while**, **if**/**else**, and **switch**/**when** statements.

As in YAML, `on` and `yes` are the same as boolean `true`, while `off` and `no` are boolean `false`.

For single-line statements, `unless` can be used as the inverse of `if`.

As a shortcut for `this.property`, you can use `@property`.

You can use `in` to test for array presence, and `of` to test for JavaScript object-key presence.

All together now:

CoffeeScript	JavaScript
`is`	`===`
`isnt`	`!==`
`not`	`!`
`and`	`&&`
`or`	`||`
`true, yes, on`	`true`
`false, no, off`	`false`
`@, this`	`this`
`of`	`in`
`in`	*no JS equivalent*

```
launch() if ignition is on

volume = 10 if band isnt SpinalTap

letTheWildRumpusBegin() unless answer is no

if car.speed < limit then accelerate()

winner = yes if pick in [47, 92, 13]

print inspect "My name is " + @name
```

load

```
var volume, winner;
if (ignition === true) {
  launch();
}
if (band !== SpinalTap) {
  volume = 10;
}
if (answer !== false) {
  letTheWildRumpusBegin();
}
if (car.speed < limit) {
  accelerate();
}
if (pick === 47 || pick === 92 || pick === 13) {
  winner = true;
}
print(inspect("My name is " + this.name));
```

The Existential Operator

It's a little difficult to check for the existence of a variable in JavaScript. `if (variable) ...` comes close, but fails for zero, the empty string, and false. CoffeeScript's existential operator `?` returns true unless a variable is **null** or **undefined**, which makes it analogous to Ruby's `nil?`

It can also be used for safer conditional assignment than `||=` provides, for cases where you may be handling numbers or strings.

```
solipsism = true if mind? and not world?

speed ?= 75

footprints = yeti ? "bear"
```

load

```
var footprints, solipsism;
if ((typeof mind !== "undefined" && mind !== null) && !(typeof
world !== "undefined" && world !== null)) {
  solipsism = true;
}
if (typeof speed !== "undefined" && speed !== null) {
  speed;
} else {
  speed = 75;
};
footprints = typeof yeti !== "undefined" && yeti !== null ? yeti
: "bear";
```

run: footprints

The accessor variant of the existential operator `?.` can be used to soak up null references in a chain of properties. Use it instead of the dot accessor `.` in cases where the base value may be **null** or **undefined**. If all of the properties exist then you'll get the expected result, if the chain is broken, **undefined** is returned instead of the **TypeError** that would be raised otherwise.

```
zip = lottery.drawWinner?().address?.zipcode
```

load

```
var zip, _ref;
zip = typeof lottery.drawWinner === "function" ? (_ref =
lottery.drawWinner().address) != null ? _ref.zipcode : void 0 :
void 0;
```

Soaking up nulls is similar to Ruby's andand gem, and to the safe navigation operator in Groovy.

Classes, Inheritance, and Super

JavaScript's prototypal inheritance has always been a bit of a brain-bender, with a whole family tree of libraries that provide a cleaner syntax for classical inheritance on top of JavaScript's prototypes: Base2, Prototype.js, JS.Class, etc. The libraries provide syntactic sugar, but the built-in inheritance would be completely usable if it weren't for a couple of small exceptions: it's awkward to call **super** (the prototype object's implementation of the current function), and it's awkward to correctly set the prototype chain.

Instead of repetitively attaching functions to a prototype, CoffeeScript provides a basic `class` structure that allows you to name your class, set the superclass, assign prototypal properties, and define the constructor, in a single assignable expression.

Constructor functions are named, to better support helpful stack traces.

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved " + meters + "m."

class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5

class Horse extends Animal
  move: ->
    alert "Galloping..."
```

```
var Animal, Horse, Snake, sam, tom;
var __hasProp = Object.prototype.hasOwnProperty, __extends =
function(child, parent) {
  for (var key in parent) { if (__hasProp.call(parent, key))
child[key] = parent[key]; }
  function ctor() { this.constructor = child; }
  ctor.prototype = parent.prototype;
  child.prototype = new ctor;
  child.__super__ = parent.prototype;
  return child;
};
Animal = (function() {
  function Animal(name) {
    this.name = name;
```

```
    super 45

sam = new Snake "Sammy the Python"
tom = new Horse "Tommy the Palomino"

sam.move()
tom.move()
```

load

```
  }
  Animal.prototype.move = function(meters) {
    return alert(this.name + " moved " + meters + "m.");
  };
  return Animal;
})();
Snake = (function() {
  function Snake() {
    Snake.__super__.constructor.apply(this, arguments);
  }
  __extends(Snake, Animal);
  Snake.prototype.move = function() {
    alert("Slithering...");
    return Snake.__super__.move.call(this, 5);
  };
  return Snake;
})();
Horse = (function() {
  function Horse() {
    Horse.__super__.constructor.apply(this, arguments);
  }
  __extends(Horse, Animal);
  Horse.prototype.move = function() {
    alert("Galloping...");
    return Horse.__super__.move.call(this, 45);
  };
  return Horse;
})();
sam = new Snake("Sammy the Python");
tom = new Horse("Tommy the Palomino");
sam.move();
tom.move();
```

run

If structuring your prototypes classically isn't your cup of tea, CoffeeScript provides a couple of lower-level conveniences. The `extends` operator helps with proper prototype setup, and can be used to create an inheritance chain between any pair of constructor functions; `::` gives you quick access to an object's prototype; and `super()` is converted into a call against the immediate ancestor's method of the same name.

```
String::dasherize = ->
  this.replace /_/g, "-"
```

load

```
String.prototype.dasherize = function() {
  return this.replace(/_/g, "-");
};
```

run: "one_two".dasherize()

Finally class definitions are blocks of executable code, which make for interesting metaprogramming possibilities. Because in the context of a class definition, `this` is the class object itself (the constructor function), you can assign static properties by using `@property: value`, and call functions defined in parent classes: `@attr 'title', type: 'text'`

Destructuring Assignment

To make extracting values from complex arrays and objects more convenient, CoffeeScript implements ECMAScript Harmony's proposed destructuring assignment syntax. When you assign an array or object literal to a value, CoffeeScript breaks up and matches both sides against each other, assigning the values on the right to the variables on the left. In the simplest case, it can be used for parallel assignment:

```
theBait   = 1000
theSwitch = 0

[theBait, theSwitch] = [theSwitch, theBait]
```

load

```
var theBait, theSwitch, _ref;
theBait = 1000;
theSwitch = 0;
_ref = [theSwitch, theBait], theBait = _ref[0], theSwitch =
_ref[1];
```

run: theBait

But it's also helpful for dealing with functions that return multiple values.

```
weatherReport = (location) ->
  # Make an Ajax request to fetch the weather...
```

```
var city, forecast, temp, weatherReport, _ref;
weatherReport = function(location) {
```

```
  [location, 72, "Mostly Sunny"]

[city, temp, forecast] = weatherReport "Berkeley, CA"
```

load

```
  return [location, 72, "Mostly Sunny"];
};
_ref = weatherReport("Berkeley, CA"), city = _ref[0], temp =
_ref[1], forecast = _ref[2];
```

run: forecast

Destructuring assignment can be used with any depth of array and object nesting, to help pull out deeply nested properties.

```
futurists =
  sculptor: "Umberto Boccioni"
  painter:  "Vladimir Burliuk"
  poet:
    name:   "F.T. Marinetti"
    address: [
      "Via Roma 42R"
      "Bellagio, Italy 22021"
    ]

{poet: {name, address: [street, city]}} = futurists
```

load

```
var city, futurists, name, street, _ref, _ref2;
futurists = {
  sculptor: "Umberto Boccioni",
  painter: "Vladimir Burliuk",
  poet: {
    name: "F.T. Marinetti",
    address: ["Via Roma 42R", "Bellagio, Italy 22021"]
  }
};
_ref = futurists.poet, name = _ref.name, _ref2 = _ref.address,
street = _ref2[0], city = _ref2[1];
```

run: name + " — " + street

Destructuring assignment can even be combined with splats.

```
tag = "<impossible>"

[open, contents..., close] = tag.split("")
```

load

```
var close, contents, open, tag, _i, _ref;
var __slice = Array.prototype.slice;
tag = "<impossible>";
_ref = tag.split(""), open = _ref[0], contents = 3 <=
_ref.length ? __slice.call(_ref, 1, _i = _ref.length - 1) : (_i
= 1, []), close = _ref[_i++];
```

run: contents.join("")

Function binding

In JavaScript, the `this` keyword is dynamically scoped to mean the object that the current function is attached to. If you pass a function as as callback, or attach it to a different object, the original value of `this` will be lost. If you're not familiar with this behavior, this Digital Web article gives a good overview of the quirks.

The fat arrow `=>` can be used to both define a function, and to bind it to the current value of `this`, right on the spot. This is helpful when using callback-based libraries like Prototype or jQuery, for creating iterator functions to pass to `each`, or event-handler functions to use with `bind`. Functions created with the fat arrow are able to access properties of the `this` where they're defined.

```
Account = (customer, cart) ->
  @customer = customer
  @cart = cart

  $('.shopping_cart').bind 'click', (event) =>
    @customer.purchase @cart
```

load

```
var Account;
var __bind = function(fn, me){ return function(){ return
fn.apply(me, arguments); }; };
Account = function(customer, cart) {
  this.customer = customer;
  this.cart = cart;
  return $('.shopping_cart').bind('click',
__bind(function(event) {
    return this.customer.purchase(this.cart);
  }, this));
};
```

If we had used `->` in the callback above, `@customer` would have referred to the undefined "customer" property of the DOM element, and trying to call `purchase()` on it would have raised an exception.

Embedded JavaScript

Hopefully, you'll never need to use it, but if you ever need to intersperse snippets of JavaScript within your CoffeeScript, you can use backticks to pass it straight through.

```
hi = `function() {
  return [document.title, "Hello JavaScript"].join(": ");
}`
```

load

```
var hi;
hi = function() {
  return [document.title, "Hello JavaScript"].join(": ");
};
```

run: hi()

Switch/When/Else

Switch statements in JavaScript are a bit awkward. You need to remember to **break** at the end of every **case** statement to avoid accidentally falling through to the default case. CoffeeScript prevents accidental fall-through, and can convert the `switch` into a returnable, assignable expression. The format is: `switch` condition, `when` clauses, `else` the default case.

As in Ruby, **switch** statements in CoffeeScript can take multiple values for each **when** clause. If any of the values match, the clause runs.

```
switch day
  when "Mon" then go work
  when "Tue" then go relax
  when "Thu" then go iceFishing
  when "Fri", "Sat"
    if day is bingoDay
      go bingo
      go dancing
  when "Sun" then go church
  else go work
```

load

```
switch (day) {
  case "Mon":
    go(work);
    break;
  case "Tue":
    go(relax);
    break;
  case "Thu":
    go(iceFishing);
    break;
  case "Fri":
  case "Sat":
    if (day === bingoDay) {
      go(bingo);
      go(dancing);
    }
    break;
  case "Sun":
    go(church);
    break;
  default:
    go(work);
}
```

Try/Catch/Finally

Try/catch statements are just about the same as JavaScript (although they work as expressions).

```
try
  allHellBreaksLoose()
  catsAndDogsLivingTogether()
catch error
  print error
finally
  cleanUp()
```

load

```
try {
  allHellBreaksLoose();
  catsAndDogsLivingTogether();
} catch (error) {
  print(error);
} finally {
  cleanUp();
}
```

Chained Comparisons

CoffeeScript borrows chained comparisons from Python — making it easy to test if a value falls within a certain range.

```
cholesterol = 127
```

```
var cholesterol, healthy;
```

```
healthy = 200 > cholesterol > 60
```

```
cholesterol = 127;
healthy = (200 > cholesterol && cholesterol > 60);
```

load run: healthy

String Interpolation, Heredocs, and Block Comments

Ruby-style string interpolation is included in CoffeeScript. Double-quoted strings allow for interpolated values, using `#{ ... }`, and single-quoted strings are literal.

```
author = "Wittgenstein"
quote  = "A picture is a fact. -- #{ author }"

sentence = "#{ 22 / 7 } is a decent approximation of π"
```

```
var author, quote, sentence;
author = "Wittgenstein";
quote = "A picture is a fact. -- " + author;
sentence = "" + (22 / 7) + " is a decent approximation of π";
```

load run: sentence

Multiline strings are allowed in CoffeeScript.

```
mobyDick = "Call me Ishmael. Some years ago --
 never mind how long precisely -- having little
 or no money in my purse, and nothing particular
 to interest me on shore, I thought I would sail
 about a little and see the watery part of the
 world..."
```

```
var mobyDick;
mobyDick = "Call me Ishmael. Some years ago -- never mind how
long precisely -- having little or no money in my purse, and
nothing particular to interest me on shore, I thought I would
sail about a little and see the watery part of the world...";
```

load run: mobyDick

Heredocs can be used to hold formatted or indentation-sensitive text (or, if you just don't feel like escaping quotes and apostrophes). The indentation level that begins the heredoc is maintained throughout, so you can keep it all aligned with the body of your code.

```
html = '''
       <strong>
         cup of coffeescript
       </strong>
       '''
```

```
var html;
html = '<strong>\n  cup of coffeescript\n</strong>';
```

load

Double-quoted heredocs, like double-quoted strings, allow interpolation.

Sometimes you'd like to pass a block comment through to the generated JavaScript. For example, when you need to embed a licensing header at the top of a file. Block comments, which mirror the syntax for heredocs, are preserved in the generated code.

```
###
CoffeeScript Compiler v1.1.0
Released under the MIT License
###
```

```
/*
CoffeeScript Compiler v1.1.0
Released under the MIT License
*/
```

load

Extended Regular Expressions

Similar to "heredocs" and "herecomments", CoffeeScript supports "heregexes" — extended regular expressions that ignore internal whitespace and can contain comments, after Perl's `/x` modifier, but delimited by `///`. They go a long way towards making complex regular expressions readable. To quote from the CoffeeScript source:

```
OPERATOR = /// ^ (
  ?: [-=]>             # function
   | [-+*/%<>&|^!?=]=  # compound assign / compare
```

```
var OPERATOR;
OPERATOR = /^(?:[-=]>|[-+*\/%<>&|^!?=]=|>>>=?|([-+:])\1|([&|
<>])\2=?|\?\.|\.{2,3})/;
```

```
    | >>>=?             # zero-fill right shift
    | ([-+:])\1         # doubles
    | ([&|<>])\2=?      # logic / shift
    | \?\.              # soak access
    | \.{2,3}           # range or splat
) ///
```

load

Cake, and Cakefiles

CoffeeScript includes a simple build system similar to Make and Rake. Naturally, it's called Cake, and is used for the build and test tasks for the CoffeeScript language itself. Tasks are defined in a file named `Cakefile`, and can be invoked by running `cake taskname` from within the directory. To print a list of all the tasks and options, just run `cake`.

Task definitions are written in CoffeeScript, so you can put arbitrary code in your Cakefile. Define a task with a name, a long description, and the function to invoke when the task is run. If your task takes a command-line option, you can define the option with short and long flags, and it will be made available in the `options` object. Here's a task that uses the Node.js API to rebuild CoffeeScript's parser:

```
fs = require 'fs'

option '-o', '--output [DIR]', 'directory for compiled code'

task 'build:parser', 'rebuild the Jison parser', (options) ->
  require 'jison'
  code = require('./lib/grammar').parser.generate()
  dir  = options.output or 'lib'
  fs.writeFile "#{dir}/parser.js", code
```

```
var fs;
fs = require('fs');
option('-o', '--output [DIR]', 'directory for compiled code');
task('build:parser', 'rebuild the Jison parser',
function(options) {
  var code, dir;
  require('jison');
  code = require('./lib/grammar').parser.generate();
  dir = options.output || 'lib';
  return fs.writeFile("" + dir + "/parser.js", code);
});
```

load

If you need to invoke one task before another — for example, running `build` before `test`, you can use the `invoke` function: `invoke 'build'`

"text/coffeescript" Script Tags

While it's not recommended for serious use, CoffeeScripts may be included directly within the browser using `<script type="text/coffeescript">` tags. The source includes a compressed and minified version of the compiler (Download current version here, 39k when gzipped) as `extras/coffee-script.js`. Include this file on a page with inline CoffeeScript tags, and it will compile and evaluate them in order.

In fact, the little bit of glue script that runs "Try CoffeeScript" above, as well as jQuery for the menu, is implemented in just this way. View source and look at the bottom of the page to see the example. Including the script also gives you access to `CoffeeScript.compile()` so you can pop open Firebug and try compiling some strings.

The usual caveats about CoffeeScript apply — your inline scripts will run within a closure wrapper, so if you want to expose global variables or functions, attach them to the `window` object.

Examples

- **sstephenson**'s Pow, a zero-configuration Rack server, with comprehensive annotated source.
- **frank06**'s riak-js, a Node.js client for Riak, with support for HTTP and Protocol Buffers.
- **technoweenie**'s Coffee-Resque, a port of Resque for Node.js.
- **assaf**'s Zombie.js, A headless, full-stack, faux-browser testing library for Node.js.

- **jashkenas**' Underscore.coffee, a port of the Underscore.js library of helper functions.
- **stephank**'s Orona, a remake of the Bolo tank game for modern browsers.
- **josh**'s nack, a Node.js-powered Rack server.

Resources

- Source Code
  Use `bin/coffee` to test your changes,
  `bin/cake test` to run the test suite,
  `bin/cake build` to rebuild the CoffeeScript compiler, and
  `bin/cake build:parser` to regenerate the Jison parser if you're working on the grammar.

  `git checkout lib && bin/cake build:full` is a good command to run when you're working on the core language. It'll refresh the lib directory (in case you broke something), build your altered compiler, use that to rebuild itself (a good sanity test) and then run all of the tests. If they pass, there's a good chance you've made a successful change.
- CoffeeScript Issues
  Bug reports, feature proposals, and ideas for changes to the language belong here.
- CoffeeScript Google Group
  If you'd like to ask a question, the mailing list is a good place to get help.
- The CoffeeScript Wiki
  If you've ever learned a neat CoffeeScript tip or trick, or ran into a gotcha — share it on the wiki. The wiki also serves as a directory of handy text editor extensions, web framework plugins, and general CoffeeScript build tools.
- The FAQ
  Perhaps your CoffeeScript-related question has been asked before. Check the FAQ first.

Web Chat (IRC)

Quick help and advice can usually be found in the CoffeeScript IRC room. Join `#coffeescript` on `irc.freenode.net`, or click the button below to open a webchat session on this page.

click to open #coffeescript

Change Log

1.1.0 - May 1, 2011

When running via `coffee` executable, `process.argv` and friends now report `coffee` instead of `node`. Better compatibility with **Node.js 0.4.x** module lookup changes. The output in the REPL is now colorized, like Node's is. Giving your concatenated CoffeeScripts a name when using `--join` is now mandatory. Fix for lexing compound division `/=` as a regex accidentally. All `text/coffeescript` tags should now execute in the order they're included. Fixed an issue with extended subclasses using external constructor functions. Fixed an edge-case infinite loop in `addImplicitParentheses`. Fixed exponential slowdown with long chains of function calls. Globals no longer leak into the CoffeeScript REPL. Splatted parameters are declared local to the function.

1.0.1 - Jan 31, 2011

Fixed a lexer bug with Unicode identifiers. Updated REPL for compatibility with Node.js 0.3.7. Fixed requiring relative paths in the REPL. Trailing `return` and `return undefined` are now optimized away. Stopped requiring the core Node.js `"util"` module for back-compatibility with Node.js 0.2.5. Fixed a case where a conditional `return` would cause fallthrough in a `switch` statement. Optimized empty objects in destructuring assignment.

1.0.0 - Dec 24, 2010

CoffeeScript loops no longer try to preserve block scope when functions are being generated within the loop body. Instead, you can use the `do` keyword to create a convenient closure wrapper. Added a `--nodejs` flag for passing through options directly to the `node` executable. Better behavior around the use of pure statements within expressions. Fixed inclusive slicing through `-1`, for all browsers, and splicing with arbitrary expressions as endpoints.

0.9.6 - Dec 6, 2010

The REPL now properly formats stacktraces, and stays alive through asynchronous exceptions. Using `--watch` now prints timestamps as files are compiled. Fixed some accidentally-leaking variables within plucked closure-loops. Constructors now maintain their declaration location within a class body. Dynamic object keys were removed. Nested classes are now supported. Fixes execution context for naked splatted functions. Bugfix for inversion of chained comparisons. Chained class instantiation now works properly with splats.

0.9.5 - Nov 21, 2010

0.9.5 should be considered the first release candidate for CoffeeScript 1.0. There have been a large number of internal changes since the previous release, many contributed from **satyr**'s Coco dialect of CoffeeScript. Heregexes (extended regexes) were added. Functions can now have default arguments. Class bodies are now executable code. Improved syntax errors for invalid CoffeeScript. `undefined` now works like `null`, and cannot be assigned a new value. There was a precedence change with respect to single-line comprehensions: `result = i for i in list`
used to parse as `result = (i for i in list)` by default ... it now parses as `(result = i) for i in list`.

0.9.4 - Sep 21, 2010

CoffeeScript now uses appropriately-named temporary variables, and recycles their references after use. Added `require.extensions` support for **Node.js 0.3**. Loading CoffeeScript in the browser now adds just a single `CoffeeScript` object to global scope. Fixes for implicit object and block comment edge cases.

0.9.3 - Sep 16, 2010

CoffeeScript `switch` statements now compile into JS `switch` statements — they previously compiled into `if/else` chains for JavaScript 1.3 compatibility. Soaking a function invocation is now supported. Users of the RubyMine editor should now be able to use `--watch` mode.

0.9.2 - Aug 23, 2010

Specifying the start and end of a range literal is now optional, eg. `array[3..]`. You can now say `a not instanceof b`. Fixed important bugs with nested significant and non-significant indentation (Issue #637). Added a `--require` flag that allows you to hook into the `coffee` command. Added a custom `jsl.conf` file for our preferred JavaScriptLint setup. Sped up Jison grammar compilation time by flattening rules for operations. Block comments can now be used with JavaScript-minifier-friendly syntax. Added JavaScript's compound assignment bitwise operators. Bugfixes to implicit object literals with leading number and string keys, as the subject of implicit calls, and as part of compound assignment.

0.9.1 - Aug 11, 2010

Bugfix release for **0.9.1**. Greatly improves the handling of mixed implicit objects, implicit function calls, and implicit indentation. String and regex interpolation is now strictly `#{ ... }` (Ruby style). The compiler now takes a `--require` flag, which specifies scripts to run before compilation.

0.9.0 - Aug 4, 2010

The CoffeeScript **0.9** series is considered to be a release candidate for **1.0**; let's give her a

shakedown cruise. **0.9.0** introduces a massive backwards-incompatible change: Assignment now uses `=`, and object literals use `:`, as in JavaScript. This allows us to have implicit object literals, and YAML-style object definitions. Half assignments are removed, in favor of `+=`, `or=`, and friends. Interpolation now uses a hash mark `#` instead of the dollar sign `$` — because dollar signs may be part of a valid JS identifier. Downwards range comprehensions are now safe again, and are optimized to straight for loops when created with integer endpoints. A fast, unguarded form of object comprehension was added: `for all key, value of object`. Mentioning the `super` keyword with no arguments now forwards all arguments passed to the function, as in Ruby. If you extend class `B` from parent class `A`, if `A` has an `extended` method defined, it will be called, passing in `B` — this enables static inheritance, among other things. Cleaner output for functions bound with the fat arrow. `@variables` can now be used in parameter lists, with the parameter being automatically set as a property on the object — useful in constructors and setter functions. Constructor functions can now take splats.

0.7.2 - Jul 12, 2010

Quick bugfix (right after 0.7.1) for a problem that prevented `coffee` command-line options from being parsed in some circumstances.

0.7.1 - Jul 11, 2010

Block-style comments are now passed through and printed as JavaScript block comments -- making them useful for licenses and copyright headers. Better support for running coffee scripts standalone via hashbangs. Improved syntax errors for tokens that are not in the grammar.

0.7.0 - Jun 28, 2010

Official CoffeeScript variable style is now camelCase, as in JavaScript. Reserved words are now allowed as object keys, and will be quoted for you. Range comprehensions now generate cleaner code, but you have to specify `by -1` if you'd like to iterate downward. Reporting of syntax errors is greatly improved from the previous release. Running `coffee` with no arguments now launches the REPL, with Readline support. The `<-` bind operator has been removed from CoffeeScript. The `loop` keyword was added, which is equivalent to a `while true` loop. Comprehensions that contain closures will now close over their variables, like the semantics of a `forEach`. You can now use bound function in class definitions (bound to the instance). For consistency, `a in b` is now an array presence check, and `a of b` is an object-key check. Comments are no longer passed through to the generated JavaScript.

0.6.2 - May 15, 2010

The `coffee` command will now preserve directory structure when compiling a directory full of scripts. Fixed two omissions that were preventing the CoffeeScript compiler from running live within Internet Explorer. There's now a syntax for block comments, similar in spirit to CoffeeScript's heredocs. ECMA Harmony DRY-style pattern matching is now supported, where the name of the property is the same as the name of the value: `{name, length}: func`. Pattern matching is now allowed within comprehension variables. `unless` is now allowed in block form. `until` loops were added, as the inverse of `while` loops. `switch` statements are now allowed without switch object clauses. Compatible with Node.js **v0.1.95**.

0.6.1 - Apr 12, 2010

Upgraded CoffeeScript for compatibility with the new Node.js **v0.1.90** series.

0.6.0 - Apr 3, 2010

Trailing commas are now allowed, a-la Python. Static properties may be assigned directly within class definitions, using `@property` notation.

0.5.6 - Mar 23, 2010

Interpolation can now be used within regular expressions and heredocs, as well as strings. Added the `<-` bind operator. Allowing assignment to half-expressions instead of special `||=`-style operators. The arguments object is no longer automatically converted into an array. After requiring `coffee-script`, Node.js can now directly load `.coffee` files, thanks to **registerExtension**. Multiple splats can now be used in function calls, arrays, and pattern matching.

0.5.5 - Mar 8, 2010

String interpolation, contributed by Stan Angeloff. Since `--run` has been the default since **0.5.3**, updating `--stdio` and `--eval` to run by default, pass `--compile` as well if you'd like to print the result.

0.5.4 - Mar 3, 2010

Bugfix that corrects the Node.js global constants `__filename` and `__dirname`. Tweaks for more flexible parsing of nested function literals and improperly-indented comments. Updates for the latest Node.js API.

0.5.3 - Feb 27, 2010

CoffeeScript now has a syntax for defining classes. Many of the core components (Nodes, Lexer, Rewriter, Scope, Optparse) are using them. Cakefiles can use `optparse.coffee` to define options for tasks. `--run` is now the default flag for the `coffee` command, use `--compile` to save JavaScripts. Bugfix for an ambiguity between RegExp literals and chained divisions.

0.5.2 - Feb 25, 2010

Added a compressed version of the compiler for inclusion in web pages as `extras/coffee-script.js`. It'll automatically run any script tags with type `text/coffeescript` for you. Added a `--stdio` option to the `coffee` command, for piped-in compiles.

0.5.1 - Feb 24, 2010

Improvements to null soaking with the existential operator, including soaks on indexed properties. Added conditions to `while` loops, so you can use them as filters with `when`, in the same manner as comprehensions.

0.5.0 - Feb 21, 2010

CoffeeScript 0.5.0 is a major release, While there are no language changes, the Ruby compiler has been removed in favor of a self-hosting compiler written in pure CoffeeScript.

0.3.2 - Feb 8, 2010

`@property` is now a shorthand for `this.property`.
Switched the default JavaScript engine from Narwhal to Node.js. Pass the `--narwhal` flag if you'd like to continue using it.

0.3.0 - Jan 26, 2010

CoffeeScript 0.3 includes major syntax changes:
The function symbol was changed to `->`, and the bound function symbol is now `=>`.
Parameter lists in function definitions must now be wrapped in parentheses.
Added property soaking, with the `?.` operator.
Made parentheses optional, when invoking functions with arguments.
Removed the obsolete block literal syntax.

0.2.6 - Jan 17, 2010

Added Python-style chained comparisons, the conditional existence operator `?=`, and some examples from *Beautiful Code*. Bugfixes relating to statement-to-expression conversion, arguments-to-array conversion, and the TextMate syntax highlighter.

0.2.5 - Jan 13, 2010

The conditions in switch statements can now take multiple values at once — If any of them are true, the case will run. Added the long arrow `==>`, which defines and immediately binds a function to `this`. While loops can now be used as expressions, in the same way that comprehensions can. Splats can be used within pattern matches to soak up the rest of an array.

0.2.4 - Jan 12, 2010

Added ECMAScript Harmony style destructuring assignment, for dealing with extracting values from nested arrays and objects. Added indentation-sensitive heredocs for nicely formatted strings or chunks of code.

0.2.3 - Jan 11, 2010

Axed the unsatisfactory `ino` keyword, replacing it with `of` for object comprehensions. They now look like: `for prop, value of object`.

0.2.2 - Jan 10, 2010

When performing a comprehension over an object, use `ino`, instead of `in`, which helps us generate smaller, more efficient code at compile time.
Added `::` as a shorthand for saying `.prototype.`
The "splat" symbol has been changed from a prefix asterisk `*`, to a postfix ellipsis `...`
Added JavaScript's `in` operator, empty `return` statements, and empty `while` loops.
Constructor functions that start with capital letters now include a safety check to make sure that the new instance of the object is returned.
The `extends` keyword now functions identically to `goog.inherits` in Google's Closure Library.

0.2.1 - Jan 5, 2010

Arguments objects are now converted into real arrays when referenced.

0.2.0 - Jan 5, 2010

Major release. Significant whitespace. Better statement-to-expression conversion. Splats. Splice literals. Object comprehensions. Blocks. The existential operator. Many thanks to all the folks who posted issues, with special thanks to Liam O'Connor-Davis for whitespace and expression help.

0.1.6 - Dec 27, 2009

Bugfix for running `coffee --interactive` and `--run` from outside of the CoffeeScript directory. Bugfix for nested function/if-statements.

0.1.5 - Dec 26, 2009

Array slice literals and array comprehensions can now both take Ruby-style ranges to specify the start and end. JavaScript variable declaration is now pushed up to the top of the scope, making all assignment statements into expressions. You can use `\` to escape newlines. The `coffee-script` command is now called `coffee`.

0.1.4 - Dec 25, 2009

The official CoffeeScript extension is now `.coffee` instead of `.cs`, which properly belongs to C#. Due to popular demand, you can now also use `=` to assign. Unlike JavaScript, `=` can also be used within object literals, interchangeably with `:`. Made a grammatical fix for chained function calls like `func(1)(2)(3)(4)`. Inheritance and super no longer use `__proto__`, so they should be IE-compatible now.

0.1.3 - Dec 25, 2009

The `coffee` command now includes `--interactive`, which launches an interactive CoffeeScript session, and `--run`, which directly compiles and executes a script. Both options depend on a working installation of Narwhal. The `aint` keyword has been replaced by `isnt`, which goes together a little smoother with `is`. Quoted strings are now allowed as identifiers within object literals: eg. `{"5+5": 10}`. All assignment operators now use a colon: `+:`, `-:`, `*:`, etc.

0.1.2 - Dec 24, 2009

Fixed a bug with calling `super()` through more than one level of inheritance, with the re-addition of the `extends` keyword. Added experimental Narwhal support (as a Tusk package), contributed by Tom Robinson, including **bin/cs** as a CoffeeScript REPL and interpreter. New `--no-wrap` option to suppress the safety function wrapper.

0.1.1 - Dec 24, 2009

Added `instanceof` and `typeof` as operators.

0.1.0 - Dec 24, 2009

Initial CoffeeScript release.