Project Report

The Design and Implementation of ‘Cap’: A Compile-to-JavaScript Programming Language

School of Engineering and Informatics, University of Sussex

Student: Ben Gourley
Supervisor: Martin Berger

2011-2012

Statement of Originality

This report is submitted as part requirement for the degree of Internet Computing at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed: ______________________ Date: ______________________

Acknowledgements

I would like to acknowledge the support of the following people/organisations in relation to this project:

Martin Berger Martin acted as project supervisor, offering feedback and guidance throughout the project. His knowledge in the field of programming language design was invaluable.

Des Watson Des taught the compilers course that I took in my second year; much of the learning outcomes of which were used directly in this project. Des also provided help in the early stages of my complier implementation.

Clock Ltd. Clock employed me as an intern front-end web developer. The experiences there gave me the inspiration for this project.

Cara Haines I would like to thank my girlfriend for her support during the project.

Contents

Summary

JavaScript is a powerful and ubiquitous language. Its distribution exists on practically every modern personal computing device, from smart phones to PCs. As a language, it has its fair share of flaws, and the extent of its distribution, while making it a desirable platform to work on, means that updates and improvements to the language are slow to propagate.

A language that compiles to JavaScript has the ability to run anywhere that JavaScript can, which offers the opportunity to make improvements to the language as well as being available to developers for immediate use.

This report describes the design and development of 'Cap', a language that compiles to JavaScript. This new language builds upon JavaScript's good parts, makes amends for its bad parts and brings in features from other successful languages.

A compiler for Cap was implemented—successfully creating a more clutter-free and consistent language than JavaScript. The compiler makes abstractions for the un-even surface of different JavaScript environments—old and new browsers, and server side (with Node.js)—providing a unified experience for the developer.

The end result of the project is a working compiler that successfully achieved its design goals, and is ready to be adopted by developers.

1 Introduction

1.1 Project Aims

The aim of this project is to design and implement a programming language in JavaScript. The core motivation is to expose the good underlying features of JavaScript and capitalise on its ubiquity in a unified and terse syntax, while providing a layer of abstraction for its weaknesses and idioms.

The syntax of the language should be designed, as much as possible, with the programmer in mind and *not* the compiler. It should be *uniform* and *consistent*, with as few anomalies as possible. The language should be *terse* and *expressive*, allowing the programmer to write less while achieving more.

1.2 Report Structure

This report addresses the **professional considerations** of the project. It then gives a detailed explanation of the project **background**: what JavaScript is, the motivation behind the project, and an introduction to projects with similar goals. The language specification, based on an evaluation of JavaScript's shortcomings and inspiration from other languages, is gathered and formulated in the **requirements analysis** before the actual **implementation** of the compiler and its surrounding features is described. An **evaluation** of the project in terms of achieving its goals, and its practicality for real-world applications is made before the closing remarks are made in the **conclusion**.

2 Professional Considerations

The ethical issues covered in the British Computer Society's Code of Conduct[7] and Code of Practice[8], which are relevant to this project are outlined below:

- Professional Competency and Integrity: "respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work"

  Programming languages are a source of great divide in the developer community. While staying focussed on the core aims of the project, the viewpoints and criticisms of others should be upheld and used in a constructive manner. Likewise, the opinions expressed in this project should be balanced and well informed.

- Duty to Relevant Authority: "**NOT** misrepresent or withhold information on the performance of products, systems or services. . . or take advantage of the lack of relevant knowledge or inexperience of others"

  Care should be taken that any third party projects represented in this report be presented in a fair and unbiased light. Conclusions should be drawn about such projects objectively on an evidential basis.

- Duty to Relevant Authority: "accept professional responsibility for your work"

  If any of the source code authored during this project is released in the public domain, it should be coupled with an appropriate licence so that the end user takes on all responsibility for the consequences of its use.

- Act Professionally as a Specialist: "Understand the boundaries of your specialist knowledge"

  The project supervisor should be used to obtain, or find out how to obtain, knowledge outside of the project author's specialist area.

- Act Professionally as a Specialist: "Be aware that most people within the organisation do not share your expertise; avoid technical jargon and express yourself clearly in terms they understand"

  While in all specialist areas there is a core set of domain specific language, the writing style of documents produced should be written (as much as is possible) in plain English. If a potentially unfamiliar term occurs, it should be explained or referenced upon its first use.

- Act Professionally as a Specialist: "Keep in close touch with and contribute to current developments in the specialism"

  Projects of a similar vein to this one should be tracked throughout the duration of the project. The project itself fulfils contributing to current developments.

- Manage Your Workload Efficiently: "Report any overruns to budget or timescales as they become apparent"

  Along with the other points in this section, if any doubt is raised over workload, the project supervisor will be contacted for guidance.

- When Designing New Systems: "Resist the pressure to build in-house when there may be more cost effective solutions available externally and vice versa"

  In this case the cost would be time and not money, but the point is still relevant. If a suitable component of the system is available elsewhere, its quality should be evaluated and used if shown to be acceptable.

- When Programming: "Strive to produce well-structured code that facilitates testing and maintenance"

  This point is self explanatory. Testing should be an active part of development, and so code should facilitate testing at all times.

- When Testing: "Create a test environment whereby tests can be re-run and the results are predictable" and "Plan the tests to cover as many paths through the software as possible, within the constraints of time and effort"

  An automated test suite should be used and, if possible, a code coverage tool should be used to facilitate this.

- When Writing Technical Documentation: "Set a high standard of documentation" and "Strive to keep documentation up to date"

  An automated tool should be used to generate documentation from source code comments. This prevents documentation of a system in development from going quickly out of date, so long as comments are updated at the same time as the code. Every effort should be made to maintain the comments, as outdated comments can be misleading and are less useful that no comments at all.

3 Background

At its core, JavaScript is an incredibly powerful language—it is object-oriented, objects are dynamic, there are first-class functions and it is evented. This section gives an overview of JavaScript, how it came to be, and outlines some of its features. JavaScript is not without its shortcomings, in part due to its rapid birth and stunted augmentation, so this section also describes the motivation for writing a language that improves upon it.

3.1 JavaScript

3.1.1 Overview

JavaScript is a general purpose programming language. Its main use is for scripts and applications that run on web pages inside of a browser. The scope of JavaScript is such that useful behaviour may be achieved in a couple of lines of code, for instance, showing a tooltip, or it can be a platform for large web-applications such as Gmail.

3.1.2 History and Evolution

Before the introduction of JavaScript, web pages were static. To achieve any level interaction, multiple requests had to be made to the server, upon which, whole new documents would be generated, even for the smallest change. JavaScript was conceived as an embedded scripting language for the web browser by Brendan Eich at Netscape. It provided the means of manipulating HTML documents, circumventing the need for multiple page requests, thus providing richer experiences without delay or page refreshes. This facilitated functionality like client-side form validation, animation and reaction to user input (e.g. clicks and key presses). It first shipped with the Netscape browser in 1995[34], and from that point onward, every new browser came equipped with some version of JavaScript. Now a fully-fledged programming language, it is ubiquitous; since it is the [**?**] language of the web, any device with a respectable form of web access will have a JavaScript implementation. This includes all major operating systems, and devices ranging from desktops to mobiles. Most browser vendors implement their own JavaScript engine—a piece of software that will interpret and execute JavaScript code, according to the ECMAScript[1] specifica-

[1]ECMAScript is the name given to the standardised JavaScript specification (governed by the standards body ECMA International), since 'JavaScript' is a Sun/Oracle trademark.

tion. Examples of JavaScript engines include V8 (Google Chrome), Spidermonkey (Firefox) and JavaScriptCore (Safari).

Douglas Crockford, a senior JavaScript architect at Yahoo and a thought-leader in JavaScript development, describes the language as "Lisp in C's clothing"[15]. Lisp is a popular programming language for a number of reasons, including its powerful meta-programming abilities and higher-order functions. In designing the language, Eich wanted to expose some Lisp-like power to JavaScript, but he was restricted in terms of what the syntax could look like. This was due to a Netscape marketing ploy to capitalise on the popularity of Java at the time JavaScript was conceived. It had to "look like Java only less so"[19].

While the browser remains the most widespread implementation of JavaScript, its popularity has lead to appearances elsewhere. Node.js[28], a relatively new development, is one such example—a platform designed for writing networked applications (e.g. web servers). Node makes use of Google's V8 JavaScript engine, adding support for file and network IO, meaning that JavaScript can be used to write a more traditional style of program outside of the browser. It has experienced a great deal of exposure based on its speed and ideology, and as a result has become a popular platform. Other uses of JavaScript outside of the browser include plugins for Adobe Photoshop and Dashboard Widgets for Mac OS X.

3.1.3 Features

Object-orientation

Like most modern programming languages, JavaScript is object-oriented. Objects are data-structures—containers for state and behaviour. They facilitate many established software engineering practices including: *abstraction*—hiding complexity and implementation details; *encapsulation*—privacy and logical grouping of functionality; and *modularity*—separation of distinct functionality. In JavaScript, objects are unordered sets of named properties, which can be of various data types including other objects. Everything is an object, except the primitive types: `undefined`, `null`, `boolean`, `string` and `number`. Differing from objects, these primitives have only one value and can't have properties—however, the latter three appear to have properties and can be interacted with like objects; this is because they have object wrappers and JavaScript will coerce them[16].

Sometimes, the functionality of multiple objects will overlap. In order to facilitate code reuse in such cases, object-oriented languages generally offer inheritance mechanisms. Inheritance allows an object to inherit behaviour from another object, preventing duplication of their shared functionality. There are two prevailing types of inheritance: *classical* and *prototypical*. The classical approach has the concept of a *class*, which can be thought of as a blueprint or mould for objects; it defines the properties and behaviour for a certain type of object, and it can be used to create objects of that type. Inheritance is achieved by declaring that a class 'extends' another class. The prototypical approach is conceptually simpler in that objects are not created from classes, and they may inherit directly from other objects, but its use is less widespread and therefore the techniques are less well known.

Most popular programming languages, including C++, Java, Ruby and Python use classical inheritance techniques. JavaScript is the only mainstream language the employs the prototypical method. In JavaScript, all objects have a hidden link to a prototype, which is another object whose behaviour and state they inherit and extend. JavaScript's implementation of the prototype chain is exemplified in Code Snippet 1 and illustrated in Figure 1.

Code Snippet 1 Example of JavaScript's prototypical inheritance

```
// Create an animal prototype
var Animal = Object.create(null);
Animal.species = 'Unknown species';
Animal.isAlive = true;
Animal.eat = function () {
  this.species + ' is eating';
};

// Create a tiger prototype that
// inherits from the animal prototype
var Tiger = Object.create(Animal);
Tiger.species = 'Tiger';
Tiger.pounce = function () {
  return this.species + ' pounced';
};

// Create a tiger using
// the tiger prototype
var tiger = Object.create(Tiger);

tiger.eat(); // 'Tiger is eating'
tiger.pounce(); // 'Tiger pounced'
tiger.isAlive; // true
```

An `Animal` prototype is created, followed by a `Tiger` prototype, which uses the `Animal` as its prototype. A `tiger` object is created from the `Tiger` prototype. The `tiger` object has none of its own properties, but calling the methods `eat()` and `pounce()` on it causes them to be looked up on the objects in the prototype chain. When found, the methods behave as if they did belong to the `tiger` object.

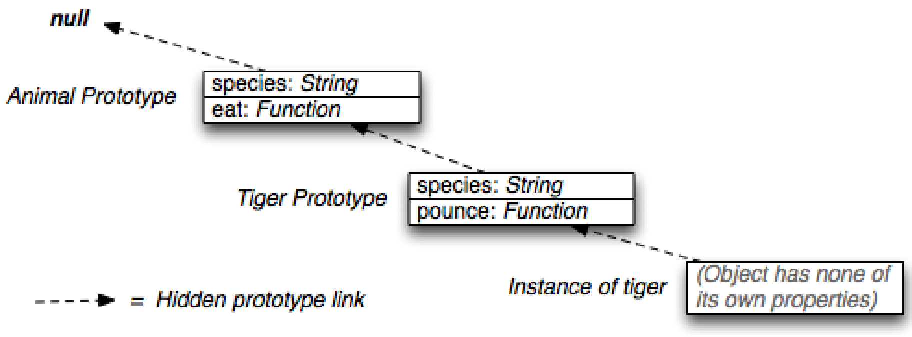


Figure 1: Illustration of JavaScript's prototypical inheritance

Dynamic Typing

Object-oriented programming languages may deal with object types in one of two different ways: *statically*—at compile-time or *dynamically*—at run-time. Compile-time checking means that type errors, for instance dividing an integer by a string, can be caught before the program is ever run, thus offering a degree of safety. As well as preventing this class of run-time error, statically typed languages generally run faster since they do not have to make the type-checks at run-time. The advantage of dynamic typing is that not all type information needs to be available at compile-time. This enables features such as *dynamic objects*—where objects can be composed and manipulated at run-time, and the `eval` *function*—where arbitrary data can be executed as code.

JavaScript is dynamically typed. It is also weakly typed, meaning that it employs various rules for implicit type conversion. This can be useful, but can also be problematic; for example, the + operator is overloaded: it is used for both addition and concatenation. If both operands are numbers, the + means addition; however, if any operand is a string, it means concatenation. Building strings is a case where this is useful:

```
'There are ' + amount + ' widgets'
```

This implicit conversion can be an issue when the programmer makes assumptions about variables' type.

First Class Functions

In JavaScript, functions are first-class citizens of the language, meaning that they can be passed around just like any other value. The support of first-class functions also means that JavaScript has higher-order functions: functions that may receive functions as arguments, or return a function. These are powerful constructs to have; providing opportunities for abstraction that result in *less* code, that is *more* expressive, and reusable[38].

JavaScript also has anonymous functions—functions that are not bound to an identifier. These are useful for defining small, use-once functions for passing as arguments.

JavaScript functions also behave as closures—they close over the scope in which they are created, meaning all of the variables in scope at that time are available inside the function when it is called. This is useful for implementing private state, as shown in Code Snippet 2.

Code Snippet 2 Privacy with closures

```
// A function which returns a
// counter, with a private member
// `num` and a public function
// `next`.
var createCounter = function () {
  var num = 0;
  return {
    next : function () {
      // This function closes
      // over `num` so it exists
      // beyond the execution of
      // the `createCounter` function
      return num++;
    }
  }
};

// Create counter
var counter = createCounter();

// Public access to `counter.next`
counter.next(); // 1
counter.next(); // 2
counter.next(); // 3

// Cannot access `counter.num`
counter.num; // undefined
```

Events

Typically, JavaScript executes in a single thread, and in a browser this thread also handles all of the user interaction (pressing keys, clicking links, etc.). It is very important, therefore, that programs do not block unnecessarily, otherwise the browser would become unresponsive. Blocking is where a part of the code occupies the CPU, allowing no other code to execute. In its single thread, JavaScript runs on an event loop. The convention of *event driven*, *asynchronous* programming with *callback* functions can give the impression of concurrency without the complexity of threads. This concept is illustrated by comparing an example that blocks, in Code Snippet 3, and an example that makes use of the event loop, in Code Snippet 4.

Code Snippet 3 Synchronous fade-in animation

```
var startTime = Date.now(),
    animationLength = 300,
    progress;

while (progress < 1) {
  progress = (Date.now() - startTime) / animationLength;
  element.style.opacity = progress;
}

console.log('Another action');
// This statement will always execute once
// the animation has completed, as the while
// loop used for the animation blocks
```

In the context of Node.js, asynchrony is equally, if not more important. The range of potentially blocking actions extends to database calls and interacting with the filesystem—actions which have a degree of latency attached to them. If Node was used to write a web server that communicated synchronously with its connecting clients, it would only be able to service one client at a time.

3.2 Project Motivation

There are two main aspects motivating this project. Firstly, the fragmentation of JavaScript environments results in the programmer only being able to rely on the 'lowest common denominator' if they want their code to be portable. Portability is especially poignant in web development, since the developer has no control over which browser is used. Secondly, the syntax of JavaScript, while a subjective matter, could arguably be greatly improved.

3.2.1 Fragmentation

Though the fifth and current version of ECMAScript describes a language with significant improvements over previous iterations, many instances of JavaScript environments, such as out-dated web browsers, are stuck with old implementations.

Code Snippet 4 Asynchronous fade-in animation

```
var startTime = Date.now(),
    animationLength = 300,
    progress;

// setTimeout takes a function to
// place on the event loop and
// the number of milliseconds in
// the future that it should execute
setTimeout(function step() {

  progress = (Date.now() - startTime) / animationLength;
  element.style.opacity = progress;

  // If the animation isn't finished, push
  // this step function onto the event loop
  // to run again at a frame-rate of 60fps.
  if (progress < 1) setTimeout(step, 1000 / 60)

}, 0);

console.log('Another action');
// This statement will always execute before
// animation has started, as the animation
// uses the event loop for each increment

setTimeout(function () {
  // This statement will execute
  // after 100ms even though the
  // animation is in progress, due
  // to the fact that the animation
  // only runs a small piece of code
  // on each iteration of the event loop
  console.log('And another');
}, 100);
```

While Eich and others work on the future iterations of ECMAScript, referred to as ‘ES.next’[18], web developers building real applications must write code according to old standards, unable to take advantage of new features and improvements without breaking compatibility. Current market share statistics show that a quarter of page views are from browsers that are only ECMAScript 3 compatible[41]—a standard which was published in 1999, over 11 years ago.

A new language that has JavaScript as a target, and makes improvements upon it, would be useful to developers immediately.

3.2.2 Syntax

While the newest standards make amends for some of the bad parts of JavaScript, it is unlikely that syntactically much will change. Not only is syntax something that characterises a language, but it is important for new versions to be backward-compatible with existing code still works. This restricts the set of changes that can safely be made.

Syntax is important to programmers because they use it every day. Good syntax is legible and expressive, enabling quick interpretation of meaning and intent. JavaScript's syntax, while familiar to programmers used to C-like syntax, can be cluttered, with lots of parentheses and symbols. The similarity to C++ and Java can also be a drawback, leading programmers to assume similarity in other aspects of the language, causing confusion when things do not work as expected.

Syntax is a contentious issue among programmers, and a language that compiles to JavaScript can give developers choice in a domain that is dominated by a single language, and therefore a single style.

3.3 Relevant Projects

There are a number of projects with similar goals that are worth noting:

Google Traceur[23] A project that compiles 'ES.next' features to current JavaScript. This enables the eager developer to use new features while not sacrificing browser compatibility. The downside of this project is that ES.next is in a constant state of change, sometimes removing functionality, which makes Traceur a volatile platform to use.

CoffeeScript [3] A language very closely based on, and implemented in, JavaScript. It brings Ruby and Python inspired syntax, while remaining tightly coupled with the underlying JavaScript. The website points out that CoffeeScript "maps one-to-one" with the compiled JavaScript. CoffeeScript has gained so much popularity that some of its features have been considered for future versions of JavaScript in ES.next[18].

Dart[27] A programming language by Google that brings static typing and classical inheritance to JavaScript-like syntax. The project currently consists of a Dart to ECMAScript 3 compiler for immediate use. In the long term Google intends to implement a Dart virtual machine in their Chrome browser, and hopes that other vendors will follow suit, therefore replacing JavaScript as the *de facto* web programming language.

Various Compilers Compilers (or backends for existing compilers) have been written for many languages to use JavaScript as a compilation target[12]. Examples include:

- OCaml
- Scheme
- Clojure
- Java
- Ruby
- Python

4 Requirements Analysis

4.1 An Analysis of JavaScript

4.1.1 Syntax

There are two prevalent styles of syntax, *C-like* and *whitespace-significant*. C-like syntax uses semi-colons to end statements and curly-braces to denote blocks. By convention, extra whitespace is used for the readability of code, but it is not required by the compiler and is therefore 'insignificant'. Languages where whitespace is 'significant' use newline characters to separate statements, and differing levels of indentation to denote block.

Contrary to other languages with functional programming characteristics, like Haskell and ML, JavaScript uses C-like syntax. The lack of significant whitespace in JavaScript means that the two pseudo-code examples in Code Snippets 5 and 6 are equivalent:

Code Snippet 5 With whitespace

```
if (test) {
  doSomething();
} else {
  doSomethingElse();
}
```

Code Snippet 6 Without whitespace

```
if(test){doSomething();}else{doSomethingElse();}
```

The first example is significantly easier to follow, and as such, any software developer or team will enforce some style including whitespace to enhance legibility. This makes the reader rely on indentation to interpret what the code is doing. If there is an error in the indentation, then the reader's interpretation of the code will be misled, as shown in Code Snippet 7.

The call to function `doSomethingElse()` looks like it would be executed regardless of whether `test` evaluates to true of false, which is not the case.

A problem specific to JavaScript, due to the existence of function expressions and object literals, is the common occurrence of the sequence `});`. The `}` closes a function expression, the `)` closes a bracketed expression or function call, and the

Code Snippet 7 Ambiguous indentation

```
if (test) {
  doSomething();
  if (testTwo) {
    doSomething();
}

doSomethingElse();

}
```

; indicates the end of a statement. Code Snippet 8 shows an example using the jQuery library to run a sequence of animations, highlighting this issue.

Code Snippet 8 Excessive repetition of the string '`});`'

```
$('#first-element').fadeIn('fast', function () {
  $('#second-element').fadeIn(fast', function () {
    $('#third-element').fadeIn('fast', function () {
      console.log('Animations finished');
    });
  });
});
```

As previously mentioned, semi-colons are used to indicate the end of a statement. The JavaScript specification has a feature for automatic semi-colon insertion, but it is best practice to explicitly use semi-colons and not to rely on this feature. Code Snippets 9 and 10 show two examples where semi-colon insertion is unhelpful.

4.1.2 Inheritance

As Douglas Crockford states, "JavaScript is conflicted in its inheritance model"[15]. The built in inheritance system is *prototypical*, which conflicts with other C-like syntax languages like Java and PHP (and other mainstream languages like Ruby) that use *classical* inheritance.

The concept of a 'class' is not at all present in the language; however, the `new` keyword, with which classes are synonymous, is. The purpose of the `new` keyword is to invoke *constructor* functions, another concept that exists in *classical* inheritance. The presence of the `new` keyword and *constructor* functions leads to confusion amongst developers, and leads some to believe that the inheritance model in JavaScript is 'broken'. Because of this, and due to the power that prototypical inheritance provides, there are many libraries simulating *classical* inheritance in

Code Snippet 9 Automatic semi-colon insertion

```
function createPerson(name, age) {

  return // A semi-colon is automatically inserted here
  {
    name : name,
    age : age,
    species : 'human',
    arms : 2,
    legs : 2
  }; // This object literal becomes an
     // unreachable expression as the
     // function will always return from
     // the line above

}

createPerson('Ben', 23); // Returns undefined
```

Code Snippet 10 Automatic semi-colon insertion

```
foo()
[1,2,3].forEach(bar)

// ...is actually interpreted as...

foo()[1, 2, 3].forEach(bar);

// Which is not a syntax error, but
// will result in a runtime error
```

JavaScript[39][13][33][35].

4.1.3 Expressiveness

JavaScript has a powerful object literal notation for defining objects on the fly, a function expression and the standard array literal. These three expressions make it possible to define simple (but not primitive) types on the fly, saving excessive use of variable declarations. A comparison of a function call with and without these expressions is shown in Code Snippets 11 and 12. Legibility could be argued either way, as the arguments are visible in place but they clutter the function call. The expressions are, however, shorter and less repetitive.

Code Snippet 11 Without expressive expressions

```
var list = new Array();
list[0] = 'a';
list[1] = 'b';
list[2] = 'c';

var options = Object.create();
options.direction = 'reverse';
options.type = 'letters';

function callback() {
  console.log('Finished');
}

sort(list, options, callback);
```

Code Snippet 12 With expressive expressions

```
sort(
  ['a', 'b', 'c'],
  { direction : 'reverse', type : 'letters' },
  function () {
    console.log('Finished');
  }
);
```

4.1.4 Equality

In JavaScript, the true equality operator is === (triple equals). The == (double equals) does type coercion, for instance `0 == false` evaluates to `true`, and is therefore considered dangerous.

4.1.5 Scope

Scope is the context within which variables exist and can be referenced. Where other C-like languages have block-scoped variables, JavaScript has function-scope. This means that a variable defined anywhere within a function can be referenced from anywhere else in the same function, even if it was defined in a block (e.g if/else, while) inside that function.

Global scope, where variables are accessible from anywhere in the program, can be a source of unreliability and insecurity[14]; they are a potential source of namespace collision, and because they are accessible and mutable from anywhere, they are unsafe to rely on. Unfortunately, JavaScript has a *global scope*[2]. Because

[2]The default scope in Node.js is actually a 'module' scope. While safer than the global scope,

of this, measures have to be taken to avoid it. It is best practice to wrap all code in an anonymous, self-executing functions, which binds variable declarations to the scope of the function instead of the global scope. Variable declarations start with the `var` keyword. If this is omitted, the variable is assumed to be global. Code Snippet 13 illustrates these global scope issues.

Code Snippet 13 Avoiding global scope

```
var x = 10;

console.log(x); // Outputs `10' to the console

(function () {
  var y = 20;
  console.log(y); // Outputs `20'
}());

console.log(y); // Outputs `undefined' because
                // y is bound to the context
                // of the anonymous function

(function () {
  x = 30; // 'var' missing, x is assumed to be global
}());

console.log(x); // Outputs '30'. Previous
                // value is overwritten.
```

4.2 Target Users

The demographic for users of the language need not come from a JavaScript background. The language should be easier to learn than JavaScript, since it aims to be more uniform and improve on JavaScript's shortcomings. Therefore, it should be a suitable language for novice and expert programmers alike.

4.3 Inspiration from Other Languages

The *offside rule* (where significant whitespace is used to denote blocks with indentation, instead of matching parentheses), as seen in languages like Python and CoffeeScript is useful for disposing of unnecessary characters. If the indentation is required by the language, there can be no variation in style between developers, meaning code is also more portable between development teams who might otherwise have different coding styles.

since it is local to each source file, the same dangers exist.

In other functional programming languages such as Haskell and ML, functions are invoked not with parentheses, but simply by juxtaposing an argument. This does away with visual clutter, leaving only the important tokens from which to extract meaning from.

Cap is unquestionably significantly inspired by JavaScript, and by building on top of it, it is able to take advantage of its good features.

4.4 JavaScript as a Target Language

4.4.1 Compilation

In order to use JavaScript as a target language, meaning that compiled Cap programs are able run anywhere that JavaScript can, there needs to be a way of translating from one language to the other. A program that translates source code of one language to that of another is known as a **compiler**.

The typical compiler structure consists of four different components, which are illustrated in Figure 2. These broadly fall in to two different phases: analysis of the source program—lexical, syntax and semantic analysis, and synthesis of the target program—code generation.

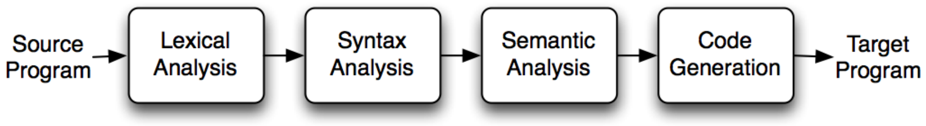


Figure 2: Typical compiler structure

The task of the **lexical analyser** is to tokenise the input—recognising characters and grouping them into basic syntactic components. The **syntax analyser**, also known as a **parser**, takes the tokens provided by the lexical analyser and matches their appearance against a set of rules about the syntax of the language. If these checks pass, and source program is syntactically valid, the parser returns an abstract representation of the program known as a ‘parse tree’ or ‘abstract syntax tree’. The **semantic analyser** generally deals with further analysis of the source program, like scope and type checking. This step may also facilitate features that exist in the source language but not in the target language (e.g. storage allocation). Finally, the **code generator** takes the output from the previous steps and generates source code of the target language.

4.4.2 Debugging

When using a high-level language as a compilation target, programs can become difficult to debug. A typical language has a stack of development resources, including tools like a debugger that can step through lines of source code. These tools will be significantly less useful when they are operating on compiled code, since the code is a step removed from what the developer has written and is likely unrecognisable.

Modern web browsers implement tools like the step-through debugger for JavaScript, and there is an open bug ticket[20] at Mozilla to get Firefox to add support for source maps. Source maps are tables that can be generated by a compiler, mapping locations in compiled code onto the locations of source where they were authored. This means that the debugger can step through JavaScript execution, but can show the developer the source of the language that they used to code with.

A rudimentary solution for debugging until browsers implement source maps would be to output the line number of the source code into a comment on the line of generated source.

4.5 Specification

Based on the analysis of JavaScript and the identification of concepts to draw in from other languages, a specification of the high-level goals of the Cap language and its compiler can now be arranged:

The language should:

- be as uniform as possible
- be as simple as possible
- have as few parentheses and unnecessary characters as possible
- use prototypical inheritance
- be dynamically typed
- have expressive literals
- use significant whitespace as a block delimiter
- have method invocation by juxtaposition
- not require that variables are declared using a keyword like `var`, instead they should be allocated when they are first used

- not require semi-colons, nor automatically insert them as a statement terminator

The compiler should:

- wrap all compiled code in an anonymous function to prevent polluting the global (or module) scope
- emit useful error messages when a compilation fails
- produce compiled code that **always** has semi-colons at the ends of statements (i.e not rely on automatic semi-colon insertion)
- produce compiled code that runs in ECMAScript 3+ environments
- provide an option to output the line number of source code as comments in the compiled source

4.6 Approach and Methodology

The evaluation of new programming languages is a difficult and unsolved problem. The only effective way to evaluate a programming language is for it to be used by many developers for many different applications—something that does not happen without the language gaining popularity. Ultimately, there is an absence of good methodologies to evaluate new programming languages; a problem which all language designers encounter. In the absence of methodologies, I will have to rely on my intuitions gained from experiences using other programming languages. Though there is no reliable way to measure the quality of the language, the success of the project can be measured by the degree in which the specification is met.

A problem exists in a language's early phase: in order to try out the language by writing programs, the compiler must work, and in order for the compiler to work, the grammar must be well-defined. This is problematic when the grammar is in a state of continuous change. To overcome this problem, the only suitable approach is to rapidly iterate and prototype. For this reason, the software engineering methodology that suits programming language design is iterative and incremental development.

Three main phases of the project were identified:

- Phase 1: Core Compiler, Autumn Term
- Phase 2: Language Features and Compiler Extensions, Autumn Term–Spring Term

- Phase 3: Proof of concept application, Spring Term[3]

The work at each of these phases was not intended to be mutually exclusive, but to give targets for the completion of large bodies of work; for example, writing the first actual application in the language in Phase 3 is likely to uncover bugs in the compiler, in which case, code from Phase 1 or 2 must be revisited.

The final design of the syntax was tightly coupled with the implementation of the compiler, so it was an underlying goal of Phases 1 and 2. While most decisions were made at these stages, finer points were continually refined throughout the duration of the project.

[3]The proof of concept application does not feature in this report—it is intended for use in the presentation.

5 Design and Implementation

5.1 Syntax

Since Cap is so heavily based on JavaScript, and makes use of many of its pre-existing features, its syntax can succinctly be defined by its differences to JavaScript. A description of the syntax in these terms follows, and various design decisions will be justified along the way. A formal specification of the language can be found in the appendices of this document.

5.1.1 Fundamentals

Cap inherits most of JavaScript's basic constructs, some with improved syntax.

Numbers

Like JavaScript, there is only one type for numerical values—`Number`. Internally this incurs a performance overhead, but provides lower cognitive load on the programmer, as they do not need to worry about precision or type-coercion.

```
5
20.1
0.0003
10e5
```

Strings

Strings are delimited with single quotes. Single quotes inside strings are escaped with '\'. In JavaScript, double quotes are also legal delimiters, but for consistency in Cap these have been removed.

```
'hello'
'it\'s all ok'
```

Booleans

In Cap, booleans are exactly the same as in JavaScript.

```
true
false
```

Logic, arithmetic and concatenation

Cap expressions differ from JavaScript expressions in two ways. Firstly, the addition operator is +, and the concatenation operator is the :, whereas in JavaScript the + is used for both. This can lead to uncertainty about what the + might do, and silent errors, due to type-coercion on operands with different types. Secondly, JavaScript has bitwise operators. Bitwise operators are often used in other languages for a performance gain on certain operations; this gain can happen because other languages have primitive data-types represented internally as bytes. In JavaScript, there are no byte-sized internal representations, and as a result the bitwise operations are slow. This, coupled with the fact that they mask intent, means that Cap does not use them. If the Cap programmer wants to do bitwise operations, for cryptography, for example, they can easily be implemented with functions. The omission of bitwise operators means the JavaScript logical operators AND (&&) and OR (||) can be used without repetition.

```
5 + 5
10 / 2
-13

'Hello,' : ' World'

a & b
a | b
!someValue
```

5.1.2 Variables and Scope

Unlike JavaScript, there is no keyword to define a variable; instead, a variable is defined when it is first assigned to. Assigning to a variable that is already defined will modify that variable, meaning that variables declared in outer scopes cannot be obscured. This is how variables are defined and behave in Python.

```
today = 'Wednesday'
result = 4 * 5 / 3
```

In JavaScript, variables have function scope, which is considered confusing and unhelpful, so much so that in ES.next, the `var` operator is being phased out in favour of the new `let` operator, which defines a block scoped variable without breaking backwards compatibility. In Cap, all variables have block scope. If a variable needs to be initialised in an outer scope, it can be assigned to the null value, which is represented by an empty tuple:

```
toBeDefined = ()
```

Like JavaScript, objects' properties can be accessed in two ways: dot and subscript notation. The latter means that property lookup can be dynamically defined at runtime.

```
# The following are equivalent
myObject.property
myObject['property']
myObject['prop' : 'erty']
```

5.1.3 Literals

JavaScript has very powerful literals. However, these are littered with parentheses and inconsistencies. Cap literals are defined with indented block syntax and a two symbol 'keyword'.

Function

Function expressions are defined with a pair of pipes delimiting 0 or more space separated arguments. The function body is a series of 1 or more statements. Unlike JavaScript, there is no return keyword. The result of the last statement is implicitly returned, like Haskell and ML. The return statement is not a part of the Cap language.

Where JavaScript has a function statement *and* a function expression, the function expression is the only way to create functions in Cap.

```
# The square function
|x|
  x * x

# A function with no args
||
  print 'hello'
```

Array

The underlying JavaScript array is a higher abstraction than a typical array—it operates more like a list, and neither the type of elements it can hold nor its size are fixed.

The array expression is a pair of square brackets. Optionally, this is followed by an indent and 1 or more line-separated items to fill the array with.

```
# Create an empty array
[]

# Create an array with
# some initial contents
[]
  'my'
  'array'
  'of'
  'items'
```

Arrays are accessed using the subscript notation with zero-based indexing:

```
# Gets the 6th item in myArray
myArray[5]

# Sets the 3rd item in myArray
myArray[2] = 10
```

Object

The object expression is a pair of curly braces. Optionally, this is followed by an indent and 1 or more line-separated assignments to initialise the object's properties with.

```
# Creates an empty object
{}

# Create an object with
# some initial properties
{}
  width = 20
  height = 40
  color = 'red'
```

5.1.4 Function Calls

Function calls differ fundamentally from JavaScript. In Cap, a function is called by juxtaposing a function with an argument, akin to the behaviour of Haskell and ML. For example:

```
print 'Hello, world!'
```

If a function returns a function, these calls can be chained. In the following example, `logger` is assumed to be a function that takes a string argument, which determines the logging function that is returned. The returned logging function is then used straight away to log a message.

```
logger 'error' 'Something bad happened'
```

Functions in Cap are conceptually different from their JavaScript counterparts. All functions take a single argument, compared with those in JavaScript that may take zero-to-many. In order to simulate this behaviour, a function may take a tuple as an argument. That tuple may be empty, simulating no arguments, or n-ary, simulating n arguments. Tuples are parenthesised comma separated lists.

```
doSomething ()
triangle (3, 4, 5)
```

If a function returns an object that has a function as one of its properties, the call can be wrapped in parentheses and the property accessed with the dot notation:

```
(getKey 'skeleton').unlock 'all'
```

Cap introduces a `where` clause for function calls, allowing placeholders to be set within the call and defined immediately after. Invocation by juxtaposition means that literals cannot be used in place for arguments, since they span multiple lines; the `where` clause goes some way to address this shortcoming. While it ends up being more to type than in-place literals, the use of a `where` clause can better exhibit the programmer's intent, as the arguments become named and also de-clutter the actual function call. The `where` clause is used like so:

```
element.animate properties where
  properties = {}
    top = 100
    left = 200
    opacity = 1
```

5.1.5 Control Structures

Since Cap has first-class functions, almost all functionality can be achieved with functions. However, it is often more convenient to have control structures to do the most basic and common tasks.

Conditionals

Conventional `if/else` statements are indented blocks. Unlike JavaScript, the condition does not need to be wrapped in parentheses.

```
if itWorked
  print 'Hurray!'
else if itFailed
  print 'Uh oh!'
else
  print 'Who knows...'
```

Cap also offers a shorthand conditional, inspired by the ternary expression in JavaScript. Where an `if/else` would be used with only a single statement in each clause, the shorthand conditional can be used instead. The shorthand conditional consists of an expression followed by a question mark, then one or two indented statements—the first is executed if the expression evaluates to true, and the second (if it exists) if the expression evaluates to false.

```
haveErrors ?
  showErrors ()

success ?
  print 'Success!'
  print 'Uh oh!'
```

Loops

The only loop structure carried over from JavaScript is the `while` loop, since all other loops can be simulated with it. In Cap, when looping over collections, or enumerating an object, it is recommended to use a functional loop, e.g. `Array.forEach()`.

```
i = 0
while i < 10
  print i
  i += 1

list = []
  'apples'
  'bananas'
  'pears'

list.forEach printItem where
  printItem = |item, i|
        i == 0 ?
          print 'I bought some: ' : item
          print 'and some ' : item
```

Try/Catch

Exceptions are a construct for dealing with errors and exceptional circumstances during the run of a program. Exceptions are 'thrown', and they must be 'caught' to be dealt with. If an exception goes uncaught, it will cause the program to terminate.

JavaScript-like `try/catch` blocks are included in Cap, facilitating exception-handling. However, while they can be useful in synchronous code, throwing exceptions and the `try/catch` clause are useless for asynchronous code[37], since an asynchronous function that will callback at some point in the future with its result returns instantly, a thrown exception won't be caught by the surrounding `try/catch` block.

In Cap, the global `throw` function, which takes a string of the error message, can be used to throw an exception.

```
try
  riskyOperation 20
catch e
  print e.message

if number > 10
  throw 'The number is too big'
```

5.1.6 Inheritance

Prototypical

In JavaScript, each object has a hidden link to another object, known as its *prototype*. An attempt to access a non-existing property of an object results in a lookup on its prototype, which is repeated until the prototype chain is exhausted or the property is found. Cap leverages the underlying prototypical inheritance of JavaScript in a simple and uniform syntax. Different object creation methods in JavaScript can result in objects being initialised with various prototypes, but in Cap all objects are created with a link to the built in *object prototype*. To manipulate an object's prototype, the function `extend` takes an object (the desired prototype) and returns a function. The returned function takes an object to set the prototype of. The use of prototypical inheritance is demonstrated below:

```
# Create a useful object
vehicle = {}
  drive = |dist|
    print 'Driving ' : dist : ' miles'
```

```
# Create another object
car = {}
  wheels = 4

# car inherit vehicle's functionality
car = extend vehicle car

car.drive 10
# -> prints 'Driving 10 miles'

car.wheels
# -> 4

# Extend can also be used as a
# starting point for a new object
# by passing in an empty tuple
ferarri = extend car ()
ferrari.colour = 'red'
```

Traits

Cap offers an alternative inheritance mechanism known as traits. It is similar to the mechanism described by Schärli et al.[36]. Traits form an alternative form of inheritance—a means of code reuse through composable pieces of behaviour. In Cap, a trait is simply an object with a `name` property, and an `apply` function. The apply function receives the object to apply the trait to. State and private functionality can be maintained within the closure of the function, and public behaviour can be exposed by adding properties to the object. Traits are applied in a similar fashion to the way prototypes are used, with a simple function:

```
# Create a trait
driveable = {}
  name = 'driveable'
  apply = |obj|
   obj.drive = |dist|
    print 'Driving ' : dist : ' miles'

# Create an object
car = {}

# car gets the driveable trait
car = trait drivable car

car.drive 10
```

```
# -> prints 'Driving 10 miles'
```

The inclusion of traits as an alternative form of inheritance is an experiment for the language. It adds complexity, but the ability to compose objects from some standard pieces of behaviour may be useful, as it achieves multiple inheritance, and is not restricted by the linear inheritance chain. Further, the two models may successfully interact to provide an effective solution.

5.1.7 Notable Omissions

Regular Expression Literals

JavaScript's regular expression literals (Perl style regular expressions wrapped in forward slashes) have been left out in favour of uniform syntax. Regular expression objects can be created using the globally available `RegExp` function in Cap.

Block Comments

In Cap, comments are lines that begin with the # symbol. The hash may be preceded by whitespace, but it must be the first non-whitespace character on the line.

```
# This is a comment
```

There is no support for block comments, though this is not a conscious design decision but due to simplicity of implementation. It was deemed unimportant at this stage of development, but in theory it would be trivial to add support for both of JavaScript's comment styles: `/* */` and `//`.

Various keyword operators

JavaScripts offer some keyword operators whose behaviour is indistinguishable from that of functions. For this reason `typeof`, `instanceof` and `throw` are left out in favour of functions by the same name. This maintains consistency and uniformity in the Cap language.

5.2 Codebase

5.2.1 Environment and Workflow

The compiler was implemented in Node.js. The file system module is one of the extensions that Node brings to JavaScript, meaning that it was a suitable implementation environment for a compiler—a program which needs to be able to read

Module	Coverage
Lexer	96%
Token	100%
Parser	89%
Node	100%
Generators	96%
Compiler	88%

Table 1: Test coverage of the core modules

from and write to the filesystem. Given that Cap will be intentionally similar to JavaScript, that Node is a primary compilation target, and that JavaScript is the language that I know best, it made sense to write the compiler in JavaScript.

Git[11] was used for the project's source code management, and as a result, remote backups on GitHub[22] were made simple and easy. Most development was done on the master branch, since there was only one developer, but for some experimental changes and major component rewrites, feature branches were used, which were then merged once considered stable enough.

The code linting tool JSHint[30] was used to maintain a consistent coding style, and to prevent syntactically correct, but potentially unsafe code from being written; for example, implicit globals and omitted semi-colons.

Testing is important for the quality and integrity of software, especially for dynamically-typed languages. Unit tests were written using the BDD interface of the Mocha[25] test framework. While code coverage cannot be metric for the quality of the tests, a low amount of coverage can be the sign of a poorly tested codebase. For this reason, 100% coverage was aimed for. Mocha has a coverage reporter, and the code coverage documentation can be found at `http://bengourley.github.com/Cap`. A summary of coverage for the core modules is shown in Table 5.2.1.

Throughout development, if a bug was found, a test was written for it before the fix was made. This ensured that bug fixes were objective, and also helped to increase the number and quality of tests. After any functionality was added, or any bug was fixed, it was made sure that the changes integrated with the tests.

Regression tests were also written. These consisted of groups of files that should either successfully parse or fail to parse. Along with the set of unit tests, these would help bugs from being introduced via other bug fixes or feature implementations.

The following tasks were automated using GNU Make:

- generate documentation
- run tests

- build and show test coverage
- minify the browser shim

5.2.2 The Compiler

Initially Jison—a JavaScript port of the GNU bison parser generator, which is used by CoffeeScript—was used to generate the Cap parser. A limitation in the Bison grammar format meant that a custom lexical analyser had to be implemented to interface with the generated parser. Due to the use of indentation in blocks, it was not sufficient just to tokenise and accept the input; the level of indentation was required to be analysed. Further issues occurred with the Bison grammar format. Ambiguities struggled to be resolved when implementing function invocation by juxtaposition.

The goals of the programming language made it very difficult to create a parser for, with a standard parser generator, since some of the helpful things for the compiler were removed: semi-colons, parentheses and curly braces. Thus, the decision was made to write the parser by hand.

Structure

The core of the compiler is made up of three modules: the **lexer**, **parser** and **generator**. The **command line interface** provides the entry point and interface for the program, and the **compiler** module ties the whole process together. The structure of the compiler is illustrated in Figure 3.

Lexer

The lexer (lexical analyser) uses regular expressions to match tokens, removing them from the front of an input string. It makes an abstraction of the significant whitespace, emitting *indent*, *outdent* and *vwhitespace* tokens, which simplifies the task for the parser. An n-lookahead method is implemented, enabling the parser to ask for tokens that are not on the front of the input string, in order to help resolve ambiguities. However only a one token lookahead is required to parse the Cap grammar. The lexer will never fail to tokenise the input—if it encounters errors or illegal tokens, it simply creates an error token, deferring error-handling to the parser.

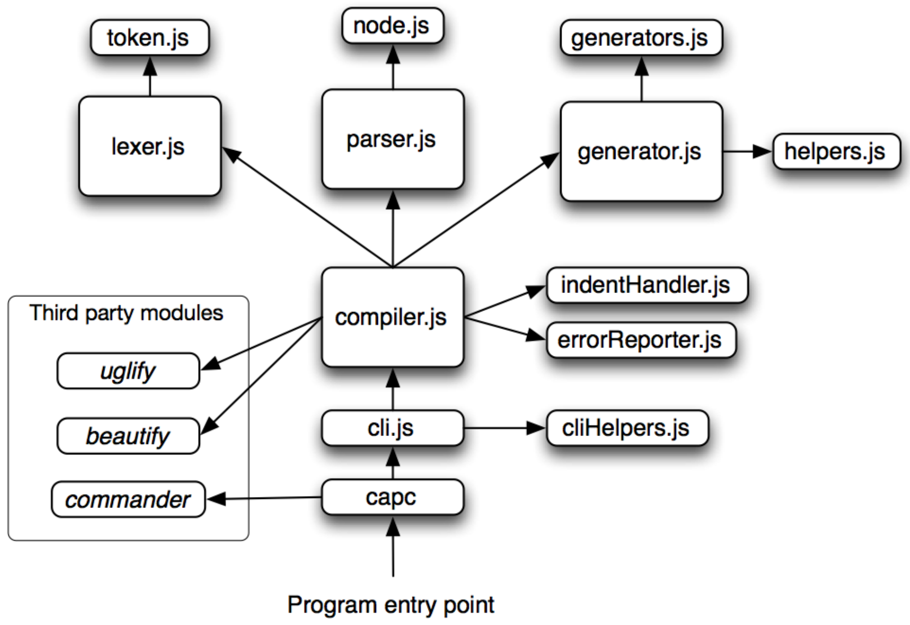


Figure 3: Structure of the Cap compiler

Parser

The parser is a top-down, left-to-right, recursive descent parser. This means that it scans the input left-to-right and has a set of recursive matching functions for the production rules of the grammar. The parser only requires a single token lookahead to disambiguate which rule it should apply, and does not require backtracking. The task of writing the parser by hand was not without its difficulties, but it was significantly more straightforward than using a parser generator.

As the parser descends the program and matches syntactic constructs of the language, it builds node objects using the **node** module. The nodes are linked together in a tree-structure, maintaining the structure of the program. When the parse finishes, the top level 'program' node is returned, providing the abstract syntax tree.

Generator

The generator orchestrates the code generation phase from the abstract syntax tree created by the parser to JavaScript. Much of the actual work is deferred to the **generators** module, which contains a generator for each type of node that could exists in the syntax tree. A program is compiled by passing a complete abstract syntax tree to the 'program' generator, whereupon all child nodes are descended and compiled recursively. Each generator function takes the node to compile and a 'meta' argument so that annotations can be made, and passed down to child nodes.

As illustrated earlier, the traditional compiler has a semantic analysis step before code generation. In the Cap compiler, due to the lack of semantic analysis required, these two steps are combined, since the checks can be made in the same pass. During the code generation, the generators maintain information about scope, and the generation will fail if an out of scope variable is used. The **helpers** module is used to define the list of predefined global variables that are in scope, and also a list of JavaScript reserved words that cannot be used as identifiers in the compiled source; the generators prefix these with an underscore to maintain legibility and validity of the JavaScript output.

As well as generating JavaScript, the generator module can also return a human readable representation of the abstract syntax tree, which is useful for understanding how the parser works, and also for debugging at this early stage in the compiler's development.

Compiler

The **compiler** module instantiates a lexer, parser, generator and an **error reporter**, and runs the compilation process. The error reporter is passed to the parser and generator upon their creation so that it can be used in the case of a parse or semantic error, halting the execution. If the compilation is successful, a string of output is returned to the compiler. Based on the options that were passed (and provided the output is JavaScript source and not a tree representation), the compiler may make use of one of the third party modules, **beatify**[32] to format or **uglify**[6] to minify and optimise the output.

Command Line Interface

The command line interface entry point is **capc**, which uses the third party module **commander**[24]: a utility for command line option parsing and automated usage information. Once the options have been parsed, the **cli** module runs and instantiates a compiler. All modules described up until this point dealt with generating a string of output—the cli deals with reading and writing files. The **cli helpers** module factors out some useful abstractions to enhance the legibility of the cli module. The cli module provides functionality for compiling a single file, or recursively scanning a directory and compiling all of the Cap source files found.

5.2.3 Features and Extensions

Browser Bundle

A primary performance concern for web developers is the number of HTTP requests that each page load incurs [40]. No matter how small the number of bytes in a file being transferred, there is a constant overhead of transporting it due to network latency. Cap takes advantage of its compilation step and bundles all of the required modules into a single file whenever it compiles for the browser. This means that regardless of the number of modules, the application is always only one file. The compiled source is 'pretty-printed' by default to help with debugging, but the compiler has the to option to minify it instead.

The browser bundle includes some extra JavaScript to provide some of the features that Cap offers. Along with the global inheritance functions, the ECMAScript 5 shim[31] is included to create the functionality in browsers that do not already have it. It does this by testing for the existence of certain functions, and creating them if they do not exist.

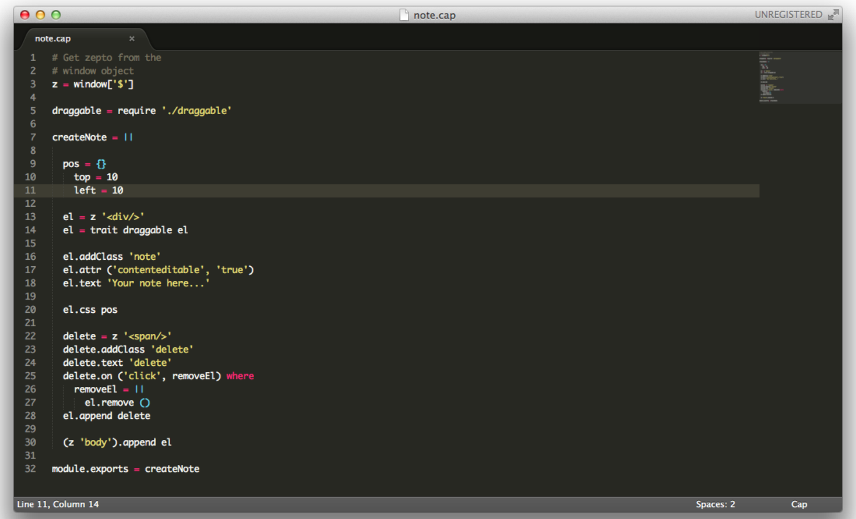
Syntax Mode for Editors

For a language to be adopted, it is necessary to have relevant development tools. One simple but important example is syntax-highlighting in text editors which enhances the readability of the code. A syntax mode was created for the editors Sublime Text 2 and Textmate, a screenshot of which can be found in Figure 4.

Module Pattern

In browser-based JavaScript development, there is no module abstraction. Application components tend to hang off of the global scope. For instance, the popular libraries jQuery[29], Underscore[4], and Raphael[5] are used by including them on the same page as some dependant code and referenced by their global variables (`$`, `_` and `Backbone` respectively). The community-driven specifications, *Asynchronous Module Definition*[21] and *CommonJS*[17] have emerged, and various libraries, for example RequireJS[9], have been written to address this lack of functionality.

Since the ECMAScript spec does not yet address modules, browsers offer nothing. However, Node.js provides the module pattern as built in structure. In Node,



```
# Get zepto from the
# window object
z = window['$']

draggable = require './draggable'

createNote = ||

  pos = {}
    top = 10
    left = 10

  el = z '<div/>'
  el = trait draggable el

  el.addClass 'note'
  el.attr ('contenteditable', 'true')
  el.text 'Your note here...'

  el.css pos

  delete = z '<span/>'
  delete.addClass 'delete'
  delete.text 'delete'
  delete.on ('click', removeEl) where
    removeEl = ||
      el.remove ()
  el.append delete

  (z 'body').append el

module.exports = createNote
```

Figure 4: Editor syntax highlighting

modules are mapped one to one with source files (i.e. a file may only contain a single module). By default, all code in a module is private and inaccessible from other modules. Functionality is made available to external modules by attaching properties to the `exports` object. An external module is brought in with the `require()` function; the result of which is the referenced module's `exports` object that can be assigned to a variable.

Not only does this implementation create the notion of privacy, it encourages modular development and the separation of functionality into reusable components. It also avoids potential namespace clashes, since the required module is just a regular javascript object, and the user (not the author) of the module can decide the name of the variable that it should occupy. This behaviour employed by Node essentially follows the commonJS module loading proposal, but with a slight extension. `exports` is a pre-existing object and can only have properties added; it cannot be re-assigned. If the module author wants to export a function, array, any other type or just a custom object, Node offers the alternative `module.exports` property which can have anything assigned to it.

The compilation step that Cap requires gives the opportunity to build-in similar functionality for the browser. Not only does this improve the standard development workflow for writing code for the browser, it also brings consistency to the Cap platform. The Cap developer does not need to decide what target environment their program will run in before making the architectural decision of whether to use modules or not. The following contrived example illustrates the usage of modules:

```
# multiplier.cap
exports.double = |num|
  num * 2

exports.triple = |num|
  num * 3
```

```
# adder.cap
module.exports = |a b|
  a + b
```

```
# script.cap
m = require './multiplier'
add = require './adder'
print (m.double 5)
print (m.triple 10)
print (add (2, 3))
```

Compiling these source files for either the browser or node and running `script.cap` would result in the following output:

```
10
30
5
```

Module loaders like require.js have a development mode, where required modules are loaded asynchronously with extra HTTP requests, and a production mode where all of the modules are bundled up and concatenated for one download. Again, since Cap always needs to be compiled, whether in development or production, the bundle is always created. This means that there is no difference between development and production code, lowering the chance of environment-specific bugs.

5.3 Distribution

The Cap compiler is available in npm (the package manager which comes with Node.js) and since Node and npm are cross platform, so too is the Cap compiler (meaning the installation process is the same, regardless of operating system). The installation is a simple one-line command in the terminal or command prompt: `npm install Cap -g`. This installs the compiler and all of its dependencies in a global location on the system (indicated by the -g argument). Once installed the

Cap compiler is available as a command line utility: `capc`. Verification that the compiler has been successfully installed and usage information can be displayed by running the command `capc -h`.

6 Evaluation

6.1 Reflection on Language Specification

This section will evaluate the success of the project in relation to the specification proposed in the Requirements Analysis section.

Uniformity

The Cap language has uniformity at the core of its design. In Javascript, the context of an assignment affects which operator is used. An assignment statement uses '=', but in the context of an object literal a ':' is used. Cap uses the '=' in both situations. Where JavaScript uses operators to achieve function-like behaviour with `throw` and `typeof`, Cap uses functions.

Simplicity

In JavaScript, there are multiple object creation patterns: literals, constructor functions and `Object.create()`. This can, at best, lead to inconsistency between developers and at worst, create confusion. Of these, Cap eliminates all but object literals.

In Cap, the concept of a function has been simplified. In JavaScript, a function appears to take `n` arguments where `n >= 0`. Functions in Cap always take one argument, which enables the means of invocation by juxtaposition and without parentheses. To simulate multiple arguments, an n-ary tuple may be passed, and for no arguments the empty tuple can be passed.

Clutter-free

The Cap grammar has successfully removed the need for: semi-colons as a line terminator; commas as an array element and object literal assignment delimiter; and replaced the need for curly braces with significant whitespace. Brackets are not required around function arguments, since functions are invoked by juxtaposition, nor are they required around the clause for a conditional, loop or catch.

An instance where Cap fails to alleviate clutter is where functions take an n-ary tuple as an argument, and when the result of a function is an object whose method is used immediately. In retrospect, the parentheses around object, array and function literals makes them quite useful for passing in place as arguments. CoffeeScript achieves a better balance in this trade-off using optional braces. However, this makes the syntax less uniform—one of the core aims of this project.

Inheritance

There is much confusion about inheritance in JavaScript, which is understandable given the number of ways in which it can be achieved, and the existence of `new` and constructor function even though it is prototype-based. Cap avoids these and implements prototypical inheritance with one simple function: `extend`.

Traits are implemented in Cap as an alternative form on inheritance. The existence of a second method goes against language goals to be simple and uniform, however, as mentioned it is an experiment which can afford to be made at this stage. It may turn out that traits are more popular and useful that prototypical inheritance, or it may turn out that they are not used at all. This remains to be seen and the language can adapt accordingly.

Typing

Like JavaScript, Cap remains dynamically and weakly typed. It was not in the goals of this project to address issues with typing; however dynamic type-checking increases the risk of run time errors (compared with static type-checking), compounding the debugging issues with Cap.

Literals

Literals are cleaner than in JavaScript, disposing of many unnecessary characters. The only drawback is that, as a result of being unable to put them in place in a function call, the *where* clause becomes overused.

Blocks

The delimitation of blocks by indentation enforces the style which is prevalent in curly-brace delimited languages anyway. The elevation of this optional style to required syntax means that developers can simply not use misleading indentation, since it results in a syntax error. The syntax is also cleaner as a result.

Function Invocation

In the initial implementation using the Jison parser generator, function invocation by argument juxtaposition proved difficult. However, this was more of a limitation of the bison grammar format than the parser generator itself. When the transition to a hand-coded recursive descent parser was made, the implementation was achieved.

Variable Declaration

In Cap, variables are created when they are first used, which removed the need for the `var` keyword. The only case where `var` is useful is to declare a variable in an outer scope, so that it can be used in a level above where it is first assigned to. In Cap, this is achieved by simply assigning to the nil value `()`), which can be considered more consistent.

6.2 Reflection on Compiler Specification

This section will evaluate the success of the project in relation to the specification proposed in the Requirements Analysis section.

Scope

Compiled Cap code uses the scope of an anonymous function to prevent things from being assigned to the global scope. The implementation also goes further, providing the same module pattern that Node.js uses in the browser. This adds a level of consistency to the development of a web application and circumvents the use of global variables in the browser.

Error Messages

Cap error messages are intended to be useful to the programmer. Ideally they will correctly identify the location of the error along with information about what caused it. There are some cases where it succeeds in this goal, and some where it does not. An example where Cap provides good error feedback is on scope checks, where an undefined variable produces the following output:

```
- Error on line 20:
`item` is not defined in the current scope

        +
        |
     16 |    options = {}
     17 |      height = 500
     18 |      width = 500
     19 |
 >   20 |  item.remove ()
```

```
        |
        +
```

However, there are some cases where the error message is completely unrelated to the cause of the error. The following example shows an error with indentation, where line 13 should be indented one level further than it is:

```
- Error on line 13:
Expecting `=` token, found `.`

        +
        |
      9 |      console.log 'destroy'
     10 |
     11 |  core.listen ('ready', start) where
     12 |    start = ||
 >   13 |    core.runScene 'start'
     14 |
     15 |  core.init options where
     16 |    options = {}
     17 |      height = 500
        |
        +
```

Work could be done on the existing parser to improve error messages. It would also be useful for the parser to have some form of error recovery so that it can report more than just the first error that is discovered. For these reasons, it might be useful to explore different parser generators with built-in techniques for dealing with errors, for example language.js[26]—a PEG parser generator in JavaScript with a focus on good error handling.

Compiled Source

The compiled source maintains a legible style and always produces valid JavaScript output. This is verified in the acceptance tests by using Uglify to parse the output, which will throw an error if the JavaScript is invalid.

ECMAScript Compatibility

Cap runs in ECMAScript 3+ environments, which means it is compatible with browsers as old as IE6. It does so without sacrificing functionality, since it makes use of the ECMAScript 5 shim[31]. The only drawback is the incurrence of a 10 kilobyte overhead in the download of the minified shim. To put this in to perspective, 10kB is roughly the size of a 200x150 pixel JPEG at 70% quality—in other words, quite a small proportion of the 1018kB total weight for the average web page[2].

Source location

The only goal that was not achieved was to print information about the Cap source location as comments in the JavaScript that it generated. In its current state, this poses a debugging issue for programs written in Cap. This is a feature that should be added to the compiler as a priority.

6.3 Performance

In order for the Cap language to succeed there should be minimal, if not zero overhead in performance when compared with plain JavaScript. The browser and Node, JavaScript's main environments, are asynchronous and IO bound domains rather than CPU bound, so the effect of any performance overhead should be minimal. However, as JavaScript's throughput capacity is already less than that of compiled languages since it is interpreted, it would be ideal to not to increase this deficit.

Benchmarks of four sample programs were made using the browser-based benchmarking tool JSPerf[10]. The results (Figure 5) show that there is only a small degradation of performance of compiled Cap versus plain JavaScript. The four benchmark programs can be found in the project source code in the `benchmarks` directory.

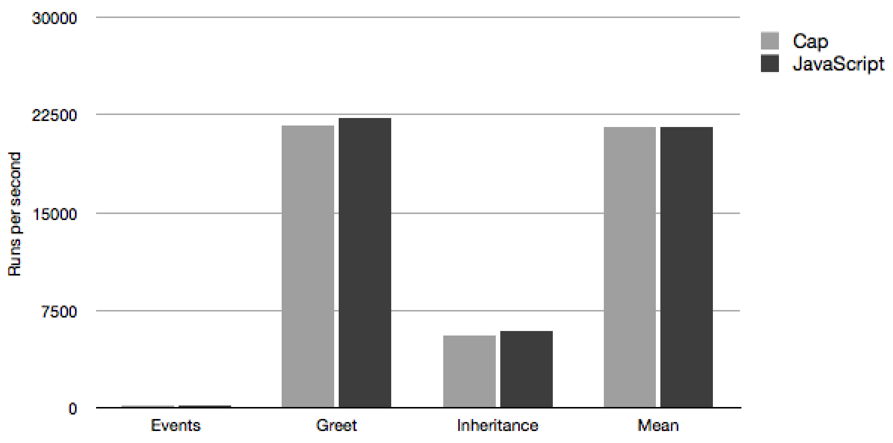


Figure 5: JSPerf benchmarks

Neither the JavaScript nor compiled Cap examples were optimised (the Cap compiler's optimisation step was not utilised), a step which may reduce the performance difference. Further tests would be needed to see if this would be the case. Performance was not a primary concern for the language at this stage of its implementation but readability of its compiled source was. These are potentially conflicting aims, but if performance became an issue, work could be done to optimise the compiled code.

6.4 Future Extensions

6.4.1 Simple Enhancements

The compiler is lacking in some simple convenience functions and extensions which would be useful and quick to add:

String Interpolation

Rather than the laborious concatenation of strings and expressions, a useful convenience is some level of string interpolation. Ruby achieves this with syntax like:

```
foobar = "blah"
"the value of foobar is #{foobar}"
```

The implementation could allow the insertion of expressions (like Ruby and CoffeeScript do) or just variables.

Operators

The compiler is missing some operators that exist in JavaScript, +=, -=, *=, /=, %. These are useful and would be relatively simple to implement.

Watch option

To ease the development workflow, a useful addition to the compiler would be a mode that watches a collection of source files, and recompiles them when they change.

6.4.2 Decoupling From JavaScript

In its current state, the Cap developer must know both Cap **and** JavaScript in order to use it. This can be problematic for newcomers, who have twice as much to learn, and problematic for JavaScript programmers, who could become confused about the boundaries between the two. However, the fact that the JavaScript APIs are transparently available, and that Cap and JavaScript objects are equivalent, mean that it is easy for Cap to interface with pre-existing JavaScript libraries. It also means that developers familiar with the JavaScript API automatically know how to achieve a lot of built-in functionality in Cap, plus they can make use of the copious amount of JavaScript libraries and frameworks that already exist.

This hybrid approach has been successful for CoffeeScript, which has seen a great deal of praise and adoption. However, CoffeeScript explicitly ties itself into JavaScript's semantics, providing a thin layer of abstraction, whereas Cap intends to be further removed. The use of the same JavaScript data structures can expose the ability to manipulate underlying objects in a way which may circumvent consciously-made design decisions of the language.

To decouple Cap from JavaScript it would be necessary to create a sort of runtime environment—a set of Cap data-types and a standard library where each of the data-types like `Object`, `Arrays` and `String` are implemented, plus built in modules like `Math`, `RegExp` and `Date`. This would mainly be a thin layer of proxies that makes use of, and extends, the functionality built into JavaScript, but it could also address any shortcomings of the data-types and libraries.

The decision to progress to the decoupled approach should not be made lightly, since the inability to interface with plain JavaScript code removes an incredibly rich resource. While making the language more secure and self-sufficient, it could render it significantly less useful. In making the transition towards this approach, a couple of techniques could be used: a transpiler could be created to create Cap source from useful JavaScript libraries, or a JavaScript interface could be created within Cap to allow the manipulation of plain JavaScript objects. However, these techniques are potentially problematic, and are an added complication for the developer. Interestingly, the designers of Scala, a language built on top of the Java Virtual Machine, decided that Java interoperability was worth the trade-offs.

One other potential drawback of the decoupling from JavaScript is the increased size of the compiled Cap source. However, this is not an issue when the target environment is Node.js, since the code does not have to be transported anywhere, and in the browser, the application would have to be of a certain scale to justify using a language other than JavaScript anyway. Since the Cap library will be of constant size, as the scale of the application grows, the overhead will become smaller.

6.4.3 Bootstrapping

For new programming languages, a compiler has to be written in some existing language. It is common for a new compiler to be re-written in the new language and have it compiled by the existing compiler. This process is known as bootstrapping[42].

Not only does this create the opportunity to write a large-scale application in the language, giving the ability to uncover potential improvements, but for Cap this would mean the compiler no longer depended on Node. This means it would be possible to use it in the browser, facilitating a browser-based interactive introduction, like CoffeeScript has, or scratch pad, similar to JSFiddle[1], where users can edit and run JavaScript in the browser.

6.4.4 User Testing

It would useful to gather some opinions on Cap by having people write programs in it. It has already been discussed that evaluating new programming languages is problematic, but some useful insight could be provided by experienced programmers, especially those with different backgrounds to myself. These opinions might ratify or conflict with the design decisions made, but if constructive they could help to evaluate and evolve the language.

7 Conclusion

Overall, the project was a great success. The design goals were achieved, and a working, feature-rich compiler was produced. More work is required before Cap can be considered ready for production, but in its current state it is ready to be released to early-adopters for alpha-testing.

The language successfully inherits the advantages of JavaScript in a clean and clear syntax, provides features where they are are missing and makes amends for JavaScripts shortcomings.

It appears that conciseness and uniformity in the design of a programming language are conflicting goals. Cap sacrifices some of its terseness for uniformity; it could be argued that it this is a good tradeoff—it makes the language easier to learn, more predictable and clearer to read, however it does mean more typing.

References

[1] jsfiddle. http://jsfiddle.net/ Last accessed: 18/04/2012.

[2] HTTP Archive. Interesting stats. http://httparchive.org/interesting.php#bytesperpage Last accessed: 17/04/2012, April 2012.

[3] Jeremy Ashkenas. Coffeescript. http://jashkenas.github.com/coffee-script/ Last accessed: 13/11/2011, .

[4] Jeremy Ashkenas. Underscore.js. http://documentcloud.github.com/underscore/ Last accessed: 15/04/2012, .

[5] Dmitry Baranovskiy. Raphaël—javascript library. http://raphaeljs.com/ Last accessed: 15/04/2012.

[6] Mihai Bazon. Uglifyjs. https://github.com/mishoo/UglifyJS Last accessed: 22/04/2012.

[7] British Computer Society. Bcs code of conduct. http://www.bcs.org/upload/pdf/conduct.pdf Last accessed: 16/11/2011, .

[8] British Computer Society. Code of good practice. http://www.bcs.org/upload/pdf/cop.pdf Last accessed: 16/11/2011, .

[9] James Burke. Requirejs. http://requirejs.org/ Last accessed: 22/04/2012.

[10] Mathias Bynens. jsperf: Javascript performance playground. http://jsperf.com/ Last accessed: 17/04/2012.

[11] Scott Chacon. Git - fast version control system. http://git-scm.com/ Last accessed: 16/11/2011.

[12] Nicolae Claudius. altjs compile-to-javascript language list. http://altjs.org/ Last accessed: 16/11/2011.

[13] Douglas Crockford. Classical inheritance in javascript. http://www.crockford.com/javascript/inheritance.html Last accessed: 16/11/2011.

[14] Douglas Crockford. http://yuiblog.com/blog/2006/06/01/global-domination/ Last accessed: 22/04/2012, June 2006.

[15] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[16] Angus Croll. The secret life of javascript primitives. `http://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/` Last accessed: 20/04/2012, September 2010.

[17] Kevin Dangoor. Commonjs modules. `http://www.commonjs.org/specs/modules/1.0/` Last accessed: 14/04/2012, 2009.

[18] Brendan Eich. Es.next. `http://www.slideshare.net/BrendanEich/esnext#text-version` Last accessed: 13/11/2011, .

[19] Brendan Eich. (untitled). `http://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1021` Last accessed: 14/11/2011, .

[20] Brendan Eich. Bug 618650 - map js source coordinates to source language that was translated to js. `https://bugzilla.mozilla.org/show_bug.cgi?id=618650` Last accessed: 16/11/2011, .

[21] Group for AMD JS Module API. amdjs-api wiki. `https://github.com/amdjs/amdjs-api/wiki` Last accessed: 22/04/2012.

[22] GitHub Inc. Github - social coding. `https://github.com/` Last accessed: 16/11/2011.

[23] Google Inc. traceur-compiler: Google's vehicle for javascript language design experimentation. `http://code.google.com/p/traceur-compiler/` Last accessed: 14/11/2011.

[24] TJ Hollowaychuk. commander.js. `https://github.com/visionmedia/commander.js/` Last accessed: 22/04/2012.

[25] TJ Hollowaychuk. Mocha - the fun, simple, flexible javascript test framework. `http://visionmedia.github.com/mocha/` Last accessed: 17/04/2012, 2011.

[26] Francisco Ryan Tolmasky I. Language.js. `http://languagejs.com/` Last accessed: 17/04/2012.

[27] Google Inc. Dart: Structured web programming. `http://www.dartlang.org/` Last accessed: 14/11/2011.

[28] Joyent, Inc. node.js. http://nodejs.org/ Last accessed: 16/11/2011.

[29] The jQuery Foundation. jquery: The write less, do more, javascript library. http://jquery.com/ Last accessed: 15/04/2012.

[30] Anton Kovalyov. Jshint, a javascript code quality tool. http://www.jshint.com/ Last accessed: 17/04/2012.

[31] Kris Kowal. Ecmascript 5 compatibility shims for legacy javascript engines. https://github.com/kriskowal/es5-shim Last accessed: 18/04/2012.

[32] Cluster Technology Limited. js-beautify-node. https://github.com/clustertech/js-beautify-node Last accessed: 22/04/2012.

[33] Peter Michaux. Class-based inheritance in javascript. http://michaux.ca/articles/class-based-inheritance-in-javascript Last accessed: 16/11/2011.

[34] Netscape Communications Corporation. Netscape and sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet. http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html Last accessed: 13/11/2011.

[35] John Resig. Simple javascript inheritance. http://ejohn.org/blog/simple-javascript-inheritance/ Last accessed: 16/11/2011.

[36] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, November 2002. URL http://scg.unibe.ch/archive/papers/Scha02bTraits.pdf. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[37] Isaac Schlueter. try/catch/throw (was: My humble co-routine proposal). https://groups.google.com/forum/#!msg/nodejs/1ESsssIxrUU/ Last accessed: 14/04/2012, November 2011.

[38] Joel Spolsky. Can your programming language do this? http://www.joelonsoftware.com/items/2006/08/01.html Last accessed: 20/04/2012, August 2006.

[39] Stoyan Stefanov. *JavaScript Patterns*. O'Reilly Media, 2010.

[40] Tenni Theurer. Performance research, part 2: Browser cache usage – exposed! `http://yuiblog.com/blog/2007/01/04/performance-research-part-2/` Last accessed: 15/04/2012, January 2007.

[41] W3Counter. W3counter - global web stats. `http://www.w3counter.com/globalstats.php?year=2011&month=10` Last accessed: 13/11/2011.

[42] Des Watson. *High-Level Languages and Their Compilers*. Addison Wesley, 1989.

Appendices

A Formal Grammar Specification

The following BNF style grammar describes Cap's syntax. Square brackets demarcate optional components.

These notational conventions are used:

- /* and */ delimit comments and character set descriptions for succinctness
- [] indicates optional components (zero or one occurrences)
- * indicates zero or many occurrences
- + indicates one or many occurrences
- | indicates choice
- () indicates grouping

```
<program>             ::= <statementList> | <empty>

<statementList>       ::= <statement>+

<statement>           ::= <singleStatement> <lineBreak>+

<singleStatement>     ::= <expression>
                        | <conditional
                        | <where>
                        | <loop>
                        | <tryCatch>

<expression>          ::= <singleLineExpression>
                        | <multiLineExpression>

<simpleExpression>    ::= <functionCall>
                        | <prefixOperator> <simpleExpression>
                        | <simpleExpression> <infixOperator>
                          <simpleExpression>
                        | <concatenation>
                        | <reference>
                        | <number>
                        | <boolean>
                        | <string>
                        | <tuple>
```

```
<complexExpression>  ::= <literal>
                       | <assignment>

<conditional>        ::= <ifelse>
                       | <shortConditional>

<ifelse>             ::= <if> <elseif>* [<else>]

<if>                 ::= "if" <simpleExpression> <lineBreak>
                           <indent> <statementList> <outdent>

<elseif>             ::= "else if" <simpleExpression> <lineBreak>
                           <indent> <statementList> <outdent>

<else>               ::= "else" <lineBreak> <indent> <statementList>
                            <outdent>

<shortConditional>   ::= <simpleExpression> "?" <lineBreak> <indent>
                           <statement> [<statement>] <outdent>

<where>              ::= <functionCall> "where" <lineBreak> <indent>
                           <assignmentList> <lineBreak> <outdent>

<loop>               ::= "while" <simpleExpression> <lineBreak>
                           <indent> <statementList> <outdent>

<trycatch>           ::= "try" <lineBreak> <indent> <statementList>
                           <outdent> "catch" <identifier> <lineBreak>
                             <indent> <statementList> <outdent>

<literal>            ::= <objectLiteral>
                       | <arrayLiteral>
                       | <functionLiteral>

<objectLiteral>      ::= "{}" [<lineBreak> <indent>
                           <assignmentList> <outdent>]

<assignmentList>     ::= <identifier> "=" <expression>
                           (<lineBreak> <identifier> "="
                             <expression>)*

<arrayLiteral>       ::= "[]" [<lineBreak> <indent>
                           <expressionList> <outdent>]

<expressionList>     ::= <expression>+
<functionLiteral>    ::= "|" <paramList> "|" [<lineBreak>
                           <indent> <statementList> <outdent>]

<paramList>          ::= <identifier>+

<functionCall>       ::= <simpleExpression> <simpleExpression>
                         /* Function calls are left associative */

<assignment>         ::= <reference> "=" <expression>

<prefixOperator>     ::= "!"
                       | "-"

<infixOperator>      ::= "+"
                       | "-"
                       | "*"
                       | "/"
                       | ">"
                       | ">="
                       | "<="
                       | "=="
                       | "!="
                       | "&"
                       | "|"
                       /* Infix operators are left associative */

<concatenation>      ::= <simpleExpression> ":" <simpleExpression>

<reference>          ::= <identifier> ("." <identifier>)*

<identifier>         ::= <identifierChar>+

<identifierChar>     ::= /* A-Z, a-z */

<number>             ::= <decimal>
                       | <scientificNotation>

<decimal>            ::= <digit>+ ["." <digit>+]

<scientificNotation> ::= <decimal> "e" <digit>+

<digit>              ::= /* The characters 0 to 9 */

<boolean>            ::= "true"
                       | "false"
```

```
<string>              ::= "'" <stringChar>* "'"

<tuple>               ::= "(" <simpleExpression>
                            ("," <simpleExpression>)* ")"

<stringChar>          ::= /* Any non-line-breaking char except "'" */

<indent>              ::= /* An indent (according to the offside
                             rule). Can be tabs or spaces */

<outdent>             ::= /* An outdent (according to the offside
                             rule). Can be tabs or spaces */

<lineBreak>           ::= /* Any single line breaking character */

<empty>               ::= /* Empty */
```

B Log

A weekly log was made of the progress achieved and short-term objectives:

August 2011: Wrote some basic idealistic syntax, sample programs and prospective grammar. Considered basic syntax changes for first iteration of compiler (simply remove colons etc.), then add syntactic sugar. Research in to JavaScript parser generators: narrowed down to PEG.js (a js implementation of Parsing Expression grammars) or Jison (a js port of Bison). Looking at relevant/similar projects, eg. CoffeeScript, Traceur, Caja. Finding relevant projects for compiler implementation esp. offside rule parsing, e.g Jade, Stylus.

3/10/11 Met with supervisor.

Progress: presented findings so far, talked about syntax

Objectives: implement parser

10/10/11 Met with supervisor.

Progress: basic parser implemented

Objectives: compile some trivial code to js

17/10/11 Met with supervisor.

Progress: Compiled some trivial js

Objectives: Create automated test suite for compiler

24/10/11 Met with supervisor.

Progress: Created automated test suite w/ code coverage report. Also found and implemented automated docs tool.

Objectives: Amend parser to use invocation by juxtaposition

31/10/11 Met with supervisor.

Progress: Amended parser to use invocation by juxtaposition. Big refactor.

Objectives: Finding difficulty avoiding ambiguity in parser generator, write parser by hand.

07/11/11 No meeting.

Progress: Foundations of parser written by hand.

Objectives: Add features to parser.

14/11/11 No meeting.

Progress: Added features to parser.

Objectives: Write interim report.

21/11/11 No meeting.

Progress: Report written and handed in.

Objectives: Away following week âĂŞ no objectives.

28/11/11 Away.

5/12/11 No meeting.

— Christmas break —

16/01/12 Met with supervisor.

Progress: None

Objectives: Complete compiler.

23/01/12 No meeting.

Progress: Progress on compiler, but realised add complexity needed.

Objectives: Tests have been lagging, catch up.

30/01/12 No meeting.

Progress: Updated tests. Moved to new test framework.

Objectives: Continue progress on compiler, factor out code generation and because a modifier needs to modify the AST

6/02/12 No meeting.

Progress: Code generation moved into generators object.

Objectives: Continue adding features to generator

13/02/12 No meeting.

Progress: Adding functionality to generators.

Objectives: Implement block scoping and 'local' to prevent overwriting vars outside scope

20/02/12 No meeting.

Progress: Implemented block scoping. Decided against 'local'. Documentation.

Objectives: Add functionality to generators. Increase test coverage

27/02/12 Met with supervisor.

Progress: Implemented block scoping. Decided against 'local'. Documentation.

Objectives: Add functionality to generators. Increase test coverage

05/03/12 No meeting.

Progress: Increased fuctionality of generators and coverage.

Objectives: None

— Easter break —

12/03/2012 No meeting.

Progress: None

Objectives: Fix some outstanding bugs.

19/03/2012 No meeting.

Progress: Bugs fixed, more documatation.

Objectives: Implement comments and array accessors

26/03/2012 No meeting.

Progress: Comments and array accessors implemented.

Objectives: Add tests for generators. Add syntax mode for editors.

02/04/2012 No meeting.

Progress: Tests added, syntax mode created.

Objectives: Features for cli.

09/04/2012 No meeting.

Progress: Features for cli. Started proof of concept app.

Objectives: Proof of concept app, bug fixes.

16/04/2012 Met with supervisor.

Progress: Finished proof of concept app.

Objectives: Finish report.