6CCS3PRJ Final Year
Lispish to JavaScript compilation

Final Project Report

Author: Daniel Marian Zurawski

Supervisor: Christian Urban

Student ID: 1015180

16th November 2012

Abstract

"The abstract is a very brief summary of the report's contents. It should be about half-a-page long. Somebody unfamiliar with your project should have a good idea of what your work is about by reading the abstract alone."...

To those familiar with JavaScript, there is a wide spread opinion that the language itself although very powerful and extensible, contains many quirks and can be difficult to master, it is however present on all of the modern Internet-enabled computers and can be used as a target language for compilers of other languages for the sole purpose of making them portable. This paper is going to define a new language named "Lispish" that is a subset of Clojure, which is a modern dialect of Lisp and it will implement a way in which the language can be compiled to JavaScript.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly
stated to the contrary.

<div align="right">

Daniel Marian Zurawski

16th November 2012

</div>

Acknowledgements

It is usual to thank those individuals who have provided particularly useful assistance, technical or otherwise, during your project. Your supervisor will obviously be pleased to be acknowledged as he or she will have invested quite a lot of time overseeing your progress.

Contents

Chapter 1

Introduction

"This is one of the most important components of the report. It should begin with a clear statement of what the project is about so that the nature and scope of the project can be understood by a lay reader. It should summarise everything that you set out to achieve, provide a clear summary of the project's background and relevance to other work, and give pointers to the remaining sections of the report, which will contain the bulk of the technical material"

....

Following the invention of high performance JavaScript compilers such as the Google V8 JavaScript Engine, raised the interest in creating programming language interpreters and compilers that target JavaScript. It enables applications written in other languages, very often high-level languages to be run on any modern web browser.

The main motivation behind this project is to investigate how compilers and interpreters for functional languages work and how a translation from a source Lisp language to a target JavaScript language can be performed. The implementation language I am planning to use is Clojure. This project offers a good opportunity to deepen understanding of functional programming using Lisp and JavaScript and how both can be used to solve complex problems in Computer Science.

I will design a small Lisp based language called Lispish and I will investigate how such language can be translated to an executable JavaScript. The language will implement a subset of Clojure programming language, which is a modern dialect of Lisp running on the JVM. As there already exists a number of similar projects that target JavaScript, I will investigate how each compares to my Lispish to JavaScript compiler

1.1 Report Structure

Chapter 2 will provide the background research and rationale behind this project. Section 2.1 of chapter 2 aims to explain the differences between functional and imperative programming paradigms. Section 2.2 goes into the details of the two main programming languages involved in the project, namely Clojure and JavaScript and tries to summarise the differences and similarities of each. Section 2.3 briefly explains the reasoning for choosing JavaScript as a target language for our Lisp languange. Section 2.4 introduces the different existing implementations of Lisp to JavaScript compilers.

Chapter 3 defines the language of Lispish and describes the compilation pipeline. Chapter 4 describes the test suite and showcases the functioning compiler using test cases as examples

List of Figures

Chapter 2

Background

The background should set the project into context by motivating the subject matter and relating it to existing published work. The background will include a critical evaluation of the existing literature in the area in which your project work is based and should lead the reader to understand how your work is motivated by and related to existing work.

This section provides a throghout background research of the domain of functional programming, Lisp and JavaScript that lead me to the rational behind Lispish design decisions.

2.1 Bridging the gap between functional and imperative paradigm

2.1.1 Functional and imperative paradigms comparison

Functional programming is a programming paradigm that differs from imperative programming in a way that it focuses solely on evaluating functions, where one input always results in the same output for a given input (referential transparency). In imperative programming, this notion is not always true, as imperative programming focuses a lot more on modyfing the state of the application as it runs. To make referential transparency, pure functional

languages try to avoid using state and mutability, by ensuring that side effects that could introduce state changes are not possible.

An example of state is preserving results in variables for later access by other parts of the program. A side effect may result from many different operations such as variable assignments, input or output operations and anything that allows two parts of the program to access the same resource at the same time.

Due to the increase of the demand for parallelisation, as more processing cores are added to modern CPUs, it is therefore essential that the software we write can be parallelised easily and without the risk of errors that could be caused by race conditions or deadlocks - which are all caused by the notion of mutability that is present in imperative languages.

The notion of pure functions may sound very impractical for a general purpose programming language, therefore functional languages used by practicioneers such as Clojure allow state, but lexicaly scoped to its own function. When state is absolutely necessary in order to improve the performance of an application or expose variable to other parts of the program, Clojure allows for so called "atoms", that improve on the classical notion of a variable, as it is still immutable, but instead an atomic swap operation of the content is performed whenever want to override the original state.

The property of immutability is also preserved for data structures, as each time a data structure is modified, a new copy of such structure is retained therefore leaving the old one in tact. This allows for much better parallelisation, as one part of the program may never modify the same data as the other part of the program, which would lead to inconsistent state.

2.2 Programming languages involved

In order to complete this project, it is necessary not only to understand the two different programming paradigms, but also the specific features of each of the languages involved - Clojure and JavaScript, as Lispish is a subset of

Clojure and the compiler itself is also programmed in Clojure.

2.2.1 Clojure

Lisp is amongst one of the worlds oldest family of programming languages, that has developed several dialects since the original Lisp was published in 1958-1960 by John McCarthy. [citation here] Lisp languages differ from other programming languages in its few original concepts, notably treating code as data, s-expressions, parenthesized Polish prefix notation and lambda expressions.

The exact expansion of the Lisp acronym is List Processing, which has its practical reasons - Lisp source code is written as lists, formally - S-Expressions [reference here]

To illustrate how a valid s-expression would look like compared to an equivalent C expression, here is an example:

```
1 == (1 * 1)
```

in C is equivalent to

```
(= 1 (* 1 1))
```

in Lisp's s-expression based prefix notation.

2.2.2 JavaScript

JavaScript is an interpreted, dynamically typed, object-oriented programming language that originated from the ECMAScript language in 1995 [REFERENCE HERE]. It was originally intended as a client side scripting language for web browser, but it has since evolved to an extent where well-known corporations such as Microsoft use it for their server side processing [REFERENCRE HERE] due to its rich support for multiple programming styles, including functional programming at its core.

JavaScript performance

The invention of the V8 Google JavaScript Engine made JavaScript stand out from other dynamic languages by making it faster than other dynamically types languages such as for e.g. Python [reference required].

Due to the fact that Lispish compiles to JavaScript, the generated code can be treated with various optimisation techniques, including the Google Closure compiler that minimises and optimises the code, by compiling the readable, yet verbose version of the JavaScript code, to a less readable but highly optimised JS code.

Portability

JavaScript interpreters are present on majority of consumer devices and are present in all of the modern web browsers. It is the basis of Rich Internet Applications and is now not only present on the front end of the web browser, but also servers as a language of choice for back ends. Most notable examples include Microsoft's cloud platform Windows Azure that operates using JavaScript both on the front end and as well as the back end, making use of the Node.Js framework for producing highly asynchronous web applications. [reference required]

2.2.3   Compiling Lispish using a dialect of Lisp

The decision to use Clojure to write a compiler for my Lisp language comes from the fact that there are large advantages of using Lisp to compile Lisp. The nature of Lisp and it's s-expressions allows us to build efficient recursive descent parsers that can take the advantage of the already present functions in our implementation language, Clojure.

Some of the typical complexities that we would encounter when trying to implement a Lisp compiler using a non-lisp imperative language such as C include having to determine if a given expression is an s-expression (list) or a symbol and then breaking the input down into its atomic form of tokens to then building a Parse Tree (ST) or an Anotated Syntax Tree (AST). In our

case, our input s-expressions with their prefix notations can be treated as a parse tree and thanks to the in-built functions, we can greatly simplify the compiler.

For example, any s-expression can be essentially type-checked using the inbuit "symbol?" or "list?" to determine if the given s-expression yields to a symbol or a list of expressions. If an input is a list, that means we have come across another s-expression and each element in the list has to be seperately evaluated.

Modern dialects of Lisp, such as Clojure, target the Java Virtual Machine making them very portable and pluggable into an existing Java applications. Other Lisp languages are very often compiled to another target language, such as C or JavaScript that can be then run on a variety of machines.

2.3   JavaScript as a target language for the Lispish language

The rationale behind selecting JavaScript as the target language is the fact that JavaScript can be executed on almost all of the Internet enabled devices, as long as they have a web browser. Percentage of JS enabled devices as of date: [insert reference here].

Our small dialect of Lisp (Lispish) language will allow generating pluggable JavaScript code. From this follows the fact that applications written in Lispish can be executed in environments where the JVM or Clojure is not present, as the generated code will be a standard JavaScript. In theory our language could even be used as a Domain Specific Language (DSL) for JavaScript applications, as long as the code would be evaluated by our compiler in a Clojure JVM environment.

JavaScript offers a great opportunity as a target language for any high-level programming language primarily due to two reasons - it's portability and performance.

2.3.1 Similarities

JavaScript is a prototype based, objected-oriented language that due to its great flexibility and full support for lambda expressions can also be classified as a functional language.

```javascript
// Attach event listener to the argument
var assign-event-listener = function(x) {
  x.addEventListener("load", function() {
    alert("All done");
  }, false)
};
```

Above example illustrates a stored function that takes a reference to a web browsers window as an argument and attaches an event listener to it. The listener then takes two arguments, a string describing the event - here "load" and the callback function - here an anonymous function that displays an alert "All done" that gets displayed after the desired event is triggered. To now show the similary between how the same expression could look like in Lisp, take for instance:

```lisp
(defn assign-event-listener [window]
  "Attach event listener to the argument"
  (window (addEventListener
            "load"
            (fn [x] (alert "All done")))))
```

Both of the expressions make use of nested functions and thus take the advantage of the lambda calculus. This abstraction can be also one-to-one mapped when peforming the compilation from a Lisp to JavaScript and thus simplyfing the compiler.

2.4 Existing Lisp to JS compilers

There already exists a number of similar projects, that each tries to solve the problem in a slightly different way, although there exists only one mature compilar that can actually generate an executable JavaScript code and it's called ClojureScript.

2.4.1 ClojureScript

ClojureScript is a Clojure to JavaScript compiler that can already generate code that can be executed in the browser and although there are examples of companies using ClojureScript for their production applications, it is difficult to operate as it requires to execute a chain of operations, including starting a JavaScript program before the Clojure code can be compiled. ClojureScript also takes the idea further and utilises Google Closure compiler to optimise the code to remove code that can be reduced, thus making it run faster, but this approach also sufers from the fact that the Closure optimising compiler very often breaks the JavaScript code that was compiled from Clojure.

2.4.2 Outlet

Outlet is a Lisp-like programming language that compiles to JavaScript. It's compilation is interesting in that the compiler itself is written in Outlet, but it is bootstraped by a JavaScript compiler. The bootstraping JavaScript compiler is written using a recursive descent parser that takes BNF -

2.4.3 LiScript

FILL IN TEXT HERE

2.4.4 clojurejs

Yet another implementation of the same concept as ClojureScript, although does not support tail recursion and lazy evaluation essentially making it a lot less appealing to the community.

Chapter 3

Design & Specification

As previously described, the project aims to create an extensible Lispish to JavaScript compiler. In order to ... we need to formalise our input language Lispish to clearly define the possible constructs that we allow in our program. As Lispish defines a subset of an existing language, it is therefore even more important to be clear on what it possible and what is not.

3.1 Designing the Lispish language

Lispish is a dynamically typed, functional language that implements a call-by-value strategy just as its superset Clojure.

The formal description of Lispish behaviour will be described using transition systems.

3.1.1 Grammar of Lispish

$F ::= \ (let\ [x\ F]\ (F))$

$|\ (if\ (F)\ F_1\ F_2)$

$|\ (defn\ name\ [args*]\ (F))$

$|\ (fn\ [arg]\ (F))$

$|\ (cond\ (F_0)\ F_1\ (F_2)\ F_3)$

$|\ T$

| X

where

$X ::= T$

$T ::= \ ()\ |\ N\ |\ B\ |\ s$

$N ::= n\ |\ (op\ N\ N)$

$B ::= b\ |\ (bop\ t1\ t2)$

$op ::= \ +\ |\ -\ |\ *\ |\ /$

$bop ::= \ >\ |\ <\ |\ =$

$s ::= \ String$

$n ::= \ Integer$

$b ::= \ true\ |\ false$

$() ::= \ List$

3.1.2 Evaluation relations (Big-Step Semantics)

Operators:

$$\text{bop}\ \frac{E_1, s \Downarrow b_1, s'\ E_2, s' \Downarrow b_2, s''}{(bop\ E_1\ E_2) \Downarrow b, s'', if(= b\ (bop\ b_1\ b_2))}$$

$$\text{op}\ \frac{E_1, s \Downarrow n_1, s'\ E_2, s' \Downarrow n_2, s''}{(op\ E_1\ E_2) \Downarrow b, s'', if(= b\ (op\ n_1\ n_2))}$$

Atomic:

$$\text{String}\ \frac{}{s \Downarrow s}$$

$$\text{Integer}\ \frac{}{n \Downarrow n}$$

$$\text{List}\ \frac{n \Downarrow v}{(n) \Downarrow v}$$

Forms (F):

$$\text{let}\ \frac{t_0 \Downarrow v}{(let\ [x\ (t_0)]\ (t_1)) \Downarrow t_1[x \mapsto v]}$$

$$\text{if true}\ \frac{t_0 \Downarrow \text{TRUE}\quad t_1 \Downarrow v}{(if\ (t_0)\ t_1\ t_2) \Downarrow v)}$$

$$\text{if false}\ \frac{t_0 \Downarrow \text{FALSE}\quad t2 \Downarrow v}{(if\ (t_0)\ t_1\ t_2) \Downarrow v)}$$

$$\text{cond}\,\frac{t_0 \Downarrow \text{FALSE} \quad t_1 \Downarrow \text{TRUE} \quad t_3 \Downarrow v}{(cond\ (t_0)\ t_2\ (t_1)\ t_3) \Downarrow v)}$$

$$\text{defn}\,\frac{t_0\ [x] \Downarrow v}{(defn\ s\ [x]\ (t_0)) \Downarrow s \mapsto v}$$

$$\text{fn}\,\frac{t_0\ [x] \Downarrow v}{(fn\ [x]\ (t_0) \Downarrow v}$$

3.2 Development methodology

In order to streamline the process of development of the compiler, I have decided to use the Test Driven Development (TDD) methodology that emphasizes on building small units of functionalities that can be invidually tested by designeted unit tests.

Clojure allows developers to create programs using the REPL (Read Evaluation Print Loop), which is characteristic feature in new dynamic programming languages. It allows you to write your functions, evaluate them and get an instant result from an interpreter that interacts with your code. This in essence reduces the amount of unit tests that have to be implemented for trivial functions in a TDD project. REPL is a great resource for rapid development and prototyping of functions, but also ensuring that they yield the right result before the project as a whole is compiled.

3.2.1 Unit tests

Even though Clojure provides REPL, it is still important to develop a regression testing suite that ensures whenever the compiler is modified, it can still compile and yield the same result for old programs. To do this, I will use clojure.test API that provides a set of macros for evaluating forms and ensuring they yield the expected result.

3.3 Compiling Lispish to JavaScript (Compiler design??)

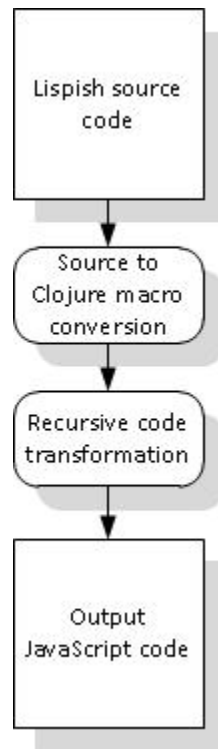### 3.3.1 Compilation pipeline (use case/state machines?)

Figure 3.1: Abstract figure of *Lispish to JavaScript* compilation.

The compiler in its simplest form will perform a one to one translation in-line translation from Lispish to JavaScript. The input source will be treated by a macro function that will prevent the code from being evaluated and it will pass it along down the pipeline to its respective emitters as ilustrated on figure **??**. Code will be treated as data and I will use the prefix notation to my advantage, treating each expression as its respective node in a parse tree.

3.3.2 Alternative approach

Chapter 4

Implementation and Testing

4.1 Translator

4.1.1 Recursive Expansion

The main idea behind the Lispish compiler implementation is recursive expansion. The compiler breaks down each s-expression it comes across into its primitives until there is no more work to be done. It then buils up the result in layers as the recursion folds upwards.

Figure **??** illustrates the functional flow chart of the compiler, without yet going deep into the details how s-expressions with multiply arity arguments are handled.

4.1.2 Forms with multiple arity (Map and reduce fns?)

In order to solve the multiple arity problem, where for instance a (cond) form can take multiple condition/true-form expression tuples and each one of them has to be individually compiled and them reduced to a string, a Map Reduce construct has been used.
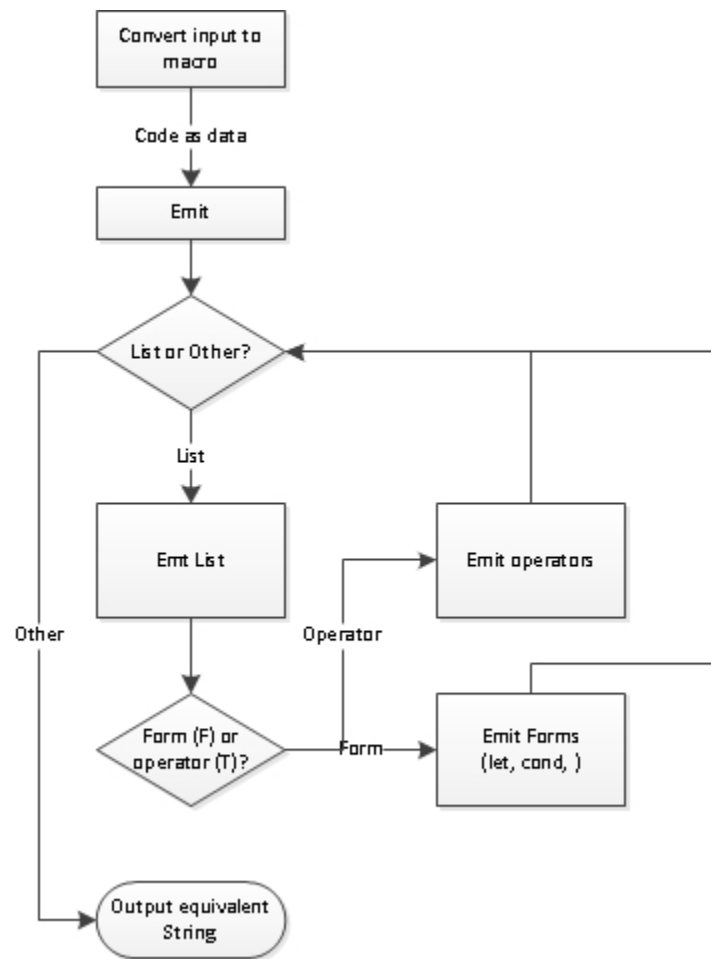
Figure 4.1: Flow chart of *Lispish to JavaScript* compilation.

Map

The idea behind the map operation is to apply a function that takes one argument, to all of the elements in a collection and return a new collection with results of each application of the aforementioned function. A simple example of Map is

```
(map (fn [x] (+ x 1))
 [0 1 2 3 4 5])
```

that yields

```
[1 2 3 4 5 6]
```

as a result

4.1.3 Reduce

Reduce generates a single result or a collection out of the application of a funcion to a collection.

```
(reduce str
 [1 2 3])
```

that yields

```
"123"
```

as a result. The collection of numbers has been reduced to a string, as each number was converted to a string and then a string of the collection has been produced. If we would to map a "str" function over the collection of [1 2 3], it would result in a new collection containing all of the elements of the old collection converted to a string, namely this list ("1" "2" "3").

To now put the map reduce constructs into perspective with Lispish, here is how a multiple arity cond (allowing practically not bound list of tests) is implemented, that not only allows for multiple arity but also generates a different, yet equivalent JavaScript code:

```
(defn emit-cond [head [name condition statement & rest]]
  (str "if(" (emit condition) ") { return " (emit statement) " }"
       (reduce str (interleave (map #(str "else if(" % ")")
                                    ;; Special case if condition is :else then emit "true"
                                    (reduce list (map #(if (= (str %) ":else") "true" (emit
                                                         (take-nth 2 rest))))
                               (map #(str "{ return " % " }")
                                    (map emit (take-nth 2 (pop rest)))))) ))
```

Given an arbitrary number of (cond (test) result) test-result tuples, we are
first emiting the first condition and the result statement. The next step is to
emit and interleave the else if and return parts of the JavaScript code, with
the exception of the form ":else" which needs to produce and "else if (true)"
and the second part of the tuple, which is the matching true form expression.
At the end of the emiting, we are reducing the interleaved emitted JavaScript
snippets to a single combined string, which results in an elegant transformation
of a Lisps (cond (test) true (test2) true2) to a JavaScript if-else-if expression.

4.2   Testing

In order to test the compiler, we need ...

Let's begin our tests by a simple arithmetic expression:

```
lispish.core> (lisp-to-js (+ 2 2))
Emit Lispish:  (+ 2 2)
Emit-list head:  + , tail:  (2 2)
Emit-op, head:  + , tail:  (2 2)
Emit Lispish:  2
Emit Lispish:  2
"(2+2)"
```

As we can see, our recursion begins with passing the Lispish source code to a
lisp-to-js macro, which then begins the recursion by invoking the initial emit

step. At first, our s-expression is of the form $(+\ 2\ 2)$, which is a list. This means that the compiler has to expand the list and emit each individual expression within it. It begins by evaluating the head of the list, which happens to be an "op" operator, in this case the "+" sign. It therefore passes the head of the previous s-expression (the "+" sign), as well as the remaining part of the expression $(2\ 2)$ to emit-op. Emit-op outputs the corresponding JavaScript by first mapping the top-level recursion emit function to each element inside of the tail list $(2\ 2)$ which reaches the bottom of the recursion in one step each and then reduces the result of this to a string concatenated with the operator in the middle. The same procedure is repeated for all of the "op", as well as "bop" expressions.

In the next sub section, we will have a look at a more complex example of generating a named function that is implemnted largely with Abstract Structural Binding.

4.2.1   Abstract Structural Binding

Abstract Sutrctural Binding in simple words means destructuring. It allows for destructuring any data structure to a corresponding argument in function parameters or a let form. For example, if we define a let as follows:

```
(let [[x1 x2] [1 2]])
```

"x1" will yield 1 and "x2" will yield 2. The same principle is true for a function. If our function accepts one parameter which is a collection:

```
(defn test [[x1 x2]] (println x1 x2))
(test [1])
```

and it destructures the first two elements of the collection to x1 x2, in the above case, x1 will yield 1 and x2 null.

Lispish uses destructuring for generating all of its forms (F):

```
lispish.core> (lisp-to-js (defn square [x] (* x x)))
Emit Lispish:  (defn square [x] (* x x))
```

```
Emit-list head:  defn , tail:  (square [x] (* x x))

Emit-forms, head:  defn , full expression:  (defn square [x] (* x x))

Emit-defn, name:  square , arg:  x , arg tail:  nil , rest:  ((* x x))

Emit Lispish:  ((* x x))

Emit Lispish:  (* x x)

Emit-list head:  * , tail:  (x x)

Emit-op, head:  * , tail:  (x x)

Emit Lispish:  x

Emit Lispish:  x

"function square(x) {(x*x)}"
```

In order to split the "defn" expression into into its respective elements, the emit-defn function that gets invoked by emit-list (after determining the head of list to be "defn") performs a structural binding of the function arguments. The bindings are then used to generate the equivalent JavaScript code.

Lets look at the signature of the emit-defn function:

```
(defn emit-defn [type [defn name [arg & more] & rest]]
  )
```

as we can see, the function takes 4 arguments and 2 optional tail arguments that can be a list of an arbitrary length. The "type" argument is simply a convinience placeholder for the head of the whole expression. The actual expression begins to bind from the [defn name [arg & more] & rest] arguments. The optional more in the arguments list of arguments allows for an arbitrary length of the named function arguments and the optional rest is for the expression that follows the named function.

This structure can be then reused to output the corresponding JavaScript as follows:

```
(str "function "
  // Name of the function
  name "("
    (if (nil? more)
```

```
    // If arguments are not a list, then output just single argument
    arg
    // Else output that list of arguments, separated by a comma
    (str arg ", " (clojure.string/join ", " more))
  )
  // Close the arguments parenthesis and begin function body
  ") {"
  // Emit function body
  (emit rest)
  // Close function body
  "}"
)
```

4.2.2   Automating tests with clojure.test API

In order to ensure that the compiler is naturally expanded and all of the of
regression tests are performed whenever a new language construct is added, I
have decided to use the Test Driven Development methodology to approach this
project. The tool to support me in the task of TDD I used was the clojure.test
API. clojure.test API [REFERENCE HERE] is a unit testing framework that
provides a set of in-built forms, particularly the "is" macro that allows to
perform boolean assertions on arbitrary expressions.

```
  (deftest factorial-example
    (is (= "function factorial(n) {if(n<2) { return 1 } else { return (n*factorial(((n-1
          (lisp-to-js (defn factorial [n] (if (< n 2) 1 (* n (factorial (- n 1)))))))))
```

.Although the compiler is not a very complexy system, it still

Chapter 5

Professional Issues

Either in a seperate section or throughout the report demonstrate that you are aware of the **Code of Conduct & Code of Good Practice** issued by the British Computer Society and have applied their principles, where appropriate, as you carried out your project.

5.1   Section Heading

Chapter 6

Results/Evaluation

6.1 Section Heading

Chapter 7

Conclusion and Future Work

The project's conclusions should list the key things that have been learnt as a consequence of engaging in your project work. For example, "The use of overloading in C++ provides a very elegant mechanism for transparent parallelisation of sequential programs", or "The overheads of linear-time n-body algorithms makes them computationally less efficient than $O(nlogn)$ algorithms for systems with less than 100000 particles". Avoid tedious personal reflections like "I learned a lot about C++ programming...", or "Simulating colliding galaxies can be real fun...". It is common to finish the report by listing ways in which the project can be taken further. This might, for example, be a plan for turning a piece of software or hardware into a marketable product, or a set of ideas for possibly turning your project into an MPhil or PhD.

Appendix A

Extra Information

A.1 Test coverage of the naive Clojure recursive-descent-parser implementation

The appendices contain information that is peripheral to the main body of the report. Information typically included in the Appendix are things like tables, proofs, graphs, test cases or any other material that would break up the theme of the text if it appeared in the body of the report. It is necessary to include your source code listings in an appendix that is separate from the body of your written report (see the information on Program Listings below).

```
(ns lispish.test.core
  (:use [lispish.core])
  (:use [clojure.test]))


(deftest plus
  (is (= "(2+2)" (lisp-to-js (+ 2 2)))))


(deftest minus
  (is (= "(2-2)" (lisp-to-js (- 2 2)))))
```

```
(deftest multiply
  (is (= "(2*2)" (lisp-to-js (* 2 2)))))


(deftest divide
  (is (= "(2/2)" (lisp-to-js (/ 2 2)))))


(deftest if-form
  (is (= "if(5>10) { return true } else { return false }" (lisp-to-js (if (> 5 10) "true" "


(deftest fn-form
  (is (= "function(x) {return (x*x)}" (lisp-to-js (fn [x] (* x x))))))


(deftest let-form
  (is (= "var x;x=5;" (lisp-to-js (let [x 5])))))


(deftest let-lambda-function
  (is (= "var x;x=function(x) {return (x*5)};" (lisp-to-js (let [x (fn [x] (* x 5))])))))


(deftest defn-form
  (is (= "function square(x) {(x*x)}" (lisp-to-js (defn square [x] (* x x))))))


(deftest fibonacci-example
  (is (= "function fib(n) {if(n<2) { return 1 } else { return (fib(((n-1)))+fib(((n-2)))) }
         (lisp-to-js (defn fib [n] (if (< n 2) 1 (+ (fib (- n 1)) (fib (- n 2)))))  ))))


(deftest factorial-example
  (is (= "function factorial(n) {if(n<2) { return 1 } else { return (n*factorial(((n-1))))
         (lisp-to-js (defn factorial [n] (if (< n 2) 1 (* n (factorial (- n 1)))))))))))


(deftest ackermann-function
  (is (= "function ackermann(m, n) {if((m==0)) { return (n+1) }else if((n==0)){ return acke
```

```
(lisp-to-js (defn ackermann [m n]
              (cond (= m 0) (+ n 1)
                    (= n 0) (ackermann (- m 1) 1)
                    :else (ackermann (- m 1) (ackermann m (- n 1)))))))))))
```

Appendix B

User Guide

B.1 Instructions

You must provide an adequate user guide for your software. The guide should provide easily understood instructions on how to use your software. A particularly useful approach is to treat the user guide as a walk-through of a typical session, or set of sessions, which collectively display all of the features of your package. Technical details of how the package works are rarely required. Keep the guide concise and simple. The extensive use of diagrams, illustrating the package in action, can often be particularly helpful. The user guide is sometimes included as a chapter in the main body of the report, but is often better included in an appendix to the main report.

Appendix C

Source Code

C.1   Instructions

Complete source code listings must be submitted as an appendix to the report. The project source codes are usually spread out over several files/units. You should try to help the reader to navigate through your source code by providing a "table of contents" (titles of these files/units and one line descriptions). The first page of the program listings folder must contain the following statement certifying the work as your own: "I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary". Your (typed) signature and the date should follow this statement.

All work on programs must stop once the code is submitted. You are required to keep safely several copies of this version of the program - one copy must be kept on the departmental disk space - and you must use one of these copies in the project examination. Your examiners may ask to see the last-modified dates of your program files, and may ask you to demonstrate that the program files you use in the project examination are identical to the program files you had stored on the departmental disk space before you submitted the project. Any attempt to demonstrate code that is not included in your submitted source listings is an attempt to cheat; any such attempt will be reported to the KCL Misconduct Committee.

You may find it easier to firstly generate a PDF of your source code using a text editor and then merge it to the end of your report. There are many free tools available that allow you to merge PDF files.