6CCS3PRJ Final Year
Lispish to JavaScript compilation

Final Project Report

Author: Daniel Marian Zurawski

Supervisor: Christian Urban

Student ID: 1015180

16th November 2012

Abstract

To those familiar with JavaScript, there is a widely 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 `Lispish`, which is going to be a subset language of Clojure, a modern dialect of Lisp. It will also propose a way in which Lispish can be translated to JavaScript.

Originality Avowal

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

Daniel Marian Zurawski

16th November 2012

Acknowledgements

I would like to thank my supervisor, Dr. Christian Urban, for steering the project in the right direction.
I would also like to thank my friend Christopher Rosset for his technical input and the lengthy conversations we had on functional programming.

Contents

Chapter 1

Introduction

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 higher level languages, to be run on any modern web browser.

1.1 Motivation

There are two categories of motivations behind this project. The first category is strictly theoretical, as it touches upon interesting topics in computer science, namely, paradigm shifts and compilation. The second is much more practical and covers the engineering aspects of the project, such as the design and implementation of a translator in a functional language and the different applications of using a higher level language to simplify program creation. In the sub sections below, I will introduce the two categories of aspects relevant to this project.

1.1.1 Theoretical

This paper is going to investigate an implementation of a translator that allows for a programming paradigm shift. The translator is going to compile a functional language to an imperative language.

Preceding the implementation of the translator, I will have to design a small Lisp based language called Lispish and I will investigate how it can be translated to an executable JavaScript. Lispish is going to implement a subset of the Clojure programming language.

1.1.2 Practical

From an engineering perspective, Lispish will provide a way to write programs in Lisp, that can execute in any modern web browser. Lispish could also allow for simple interaction with the DOM elements of web pages, as long as any arbitrary JavaScript function call can be invoked from within Lispish. Another practical aspect of this project will also involve investigating how a functional Lisp language can be used for compilation, as the implementation language that will be used to implement the compiler will be Clojure, which is a modern dialect of Lisp running on the JVM.

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.

As there already exists a number of similar projects that target JavaScript, I will investigate the different implementations in the background section of the report 2.4. I will also evaluate the complete Lispish translator implementation against the different implementations in the evaluation section of the report 5.2.

1.2 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 summarize the differences and similarities of each. Section 2.3 briefly explains the reasoning for choosing JavaScript as a target language for the `Lispish` language. Section 2.4 introduces the different existing implementations of Lisp to JavaScript compilers. Section 2.5 briefly describes the ethical and professional issues that need to be taken into account.

Chapter 3 defines the grammar of Lispish and the evaluation relations using Big-Step semantics. Section 3.3 described the compilation pipeline and the main concepts of the translator. Chapter 4 describes the test suite and showcases the functioning compiler using test cases as examples. Section 4.3 illustrates how Lispish can be used to write programs that manipulate content of a browser using native JavaScript functions and how Lispish can interact with JavaScript libraries, such as jQuery.

Throughout the paper, I will refer to the program responsible for converting the Lispish source code to Javascript both as compiler and translator, as the word compiler is not very appropriate in the case of this particular implementation. The reason for this is that the translator builds up on top of the already existing Clojure language and the tools it offers, especially the reader which takes care of parsing the input code to a typed symbols and expressions.

List of Figures

Chapter 2

Background

This section provides a throughout background research of the domain of functional programming, Lisp and JavaScript that lead me to the rational behind Lispish design decisions. It also covers existing implementations of Lisp to JavaScript translators and covers the professional issues that need to be taken into consideration in this project.

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 modifying 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 practitioners such as Clojure allow state, but lexically 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. We will use Clojure as an example of a functional language, as Lispish is a subset of Clojure and the compiler itself is also programmed in Clojure.

2.2.1 Clojure

Clojure[8] is a functional language, which is implemented as a dialect of Lisp and primarily targets the Java Virtual Machine. It can also target Microsoft's Common Language Runtime, which is the virtual machine for the .NET Framework through Clojure's sub-project **clojure-clr**[10]. It also targets JavaScript by means of `ClojureScript`[11], which is a subset of Clojure that compiles to JavaScript.

Clojure is a powerful abstraction over standard Java, which as of today does not provide lambdas and any of the functional constructs that Clojure does, including immutability and treating code as data.

Lisp

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 known as S-expressions[15][17].

To illustrate how a valid s-expression would look like compared to an equivalent Java expression, figure 2.1 illustrates an excerpt from Peter Norvig's "A Retrospective on Paradigms of AI Programming"[5]. The figure illustrates the comparison of complexity in defining a lambda function in Lisp and in Java. As we can see, Lisp is not only a lot clearer syntactically, but it is also shorter and accomplishes the same goal.

The article also shows an interesting view on the recent advancements of other languages compared to the ancient Lisp and it's a retrospective to the "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp"[14], which is Peter Norvig's book about Aritficial Intelligence, where all

> ”... Java has anonymous classes, which serve some of the purposes of closures, although in a less versatile way with a more clumsy syntax. In Lisp, we can say `(lambda (x) (f (g x)))` where in Java we would have to say
>
> ```
> new UnaryFunction() {
>   public Object execute(Object x) {
>     return (Cast x).g().f();
>   }
> }
> ```
>
> where Cast is the real type of x. This would only work with classes that observe the UnaryFunction interface (which comes from JGL and is not built-in to Java). ...”

Figure 2.1: Excerpt from Peter Norvig’s Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Comparison of a Lisp and Java lambda function declaration.[14]

of the examples have been programmed in Common Lisp.

The simplicity and conciseness Lisp syntax has been the main motivating factor to choose Clojure (being a functional language of the Lisp dialect) as the basis for the Lispish language.

Portability

Due to the fact that Clojure targets the JVM, programs written in this language can be executed in any environment where the JRE is installed by means of executing Clojure programs packaged as JAR files, given that they have been packaged to include Clojure itself.

Clojure programs can co-operate with Java applications due to its great interoperability. They can be imported into Java programs as aforementioned JAR files. Clojure can also access all of the core Java static classes/methods, making it a very powerful abstraction over Java, not only because it’s a very portable, functional language that works with immutable data structures, but also because it gives an access to the vast Java libraries and the entire JVM eco-system.

2.2.2 JavaScript

JavaScript is an interpreted, dynamically typed, object-oriented programming language that originated from the ECMAScript language in 1995. 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 support it as a language of choice for deploying web applications on their cloud-service offerings due to its rich support for multiple programming styles, including functional programming at its core. More on this below in JavaScript: Portability 2.2.2.

JavaScript performance

The invention of the V8 Google JavaScript Engine made JavaScript stand out from other dynamic languages by making it significantly faster than for e.g. Python [1] or Ruby [2].

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 adoption of `node.js` for their cloud platform Windows Azure [4], as a basis for producing highly concurrent web applications. It enables developers to write server-side applications that operate using JavaScript both on the front end and as well as the back end using a single programming language.

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 Annotated 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 inbuilt `(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 separately 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.

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 is also classified as a functional language.

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

Figure 2.2: JavaScript event listener function

Figure 2.2 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.

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

Figure 2.3: Possible Lisp equivalent of event listener function

Similarly, the equivalent expressions written in Lisp could look like on figure

2.3, which is arguably a lot more readable. Nonetheless both expressions look alike.

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 performing the compilation from a Lisp to JavaScript and thus simplifying 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 compiler that can actually generate an executable JavaScript code and it's called ClojureScript.

2.4.1 ClojureScript

ClojureScript[11] 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 suffers from the possibility in which the Closure optimising compiler could break the JavaScript code that was compiled from Clojure. However, the ClojureScript project page claims, that the JavaScript generated is compatible with the `advanced` mode of the Google Closure compiler.

ClojureScript is a production ready project and has been deployed numerous times in production environment. It is also the official

2.4.2 Outlet

Outlet[13] is a Lisp-like programming language that compiles to JavaScript. It's compilation is interesting in that the compiler itself is written in Outlet, only after it is bootstrapped by a JavaScript interpreter. The bootstrapping interpreter is implemented using grammar rules similar to Backus-Naur Form context-free grammars [7] , thus ensuring that the initial interpretation of the main constructs is correct. This approach provides a solid foundation for then implementing rest of the constructs of the Outlet language, in Outlet.

Outlet does not provide the ability to define macros, thus there is no way to dynamically extend the language without modifying the compilers source, which is a big feature of a Lisp language.

2.4.3 LiScript

LiScript[6] is again a Lisp language that compiles to JavaScript. It supports roughly 20 forms, out of which 13 forms replace the normal binary and arithmetic operations of $>$, $<$ etc. and the remaining 7 are forms such as `if`, anonymous function `fn`, iterating constructs such as `iter` and `while`.

LiScript is implemented in JavaScript and it is surprisingly lightweight. The entire implementation is around 100 lines of code, but nonetheless it can generate a readable and most importantly executable JavaScript code.

LiScript allows defining new language constructs by means of macros, a special form `defmacro`. That enables for building new forms from arbitrary strings, as input to the defmacro form first modifies the code and only then evaluates it.

2.4.4 clojurejs

ClojureJS[12] is a small subset of Clojure to JavaScript compiler. ClojureJS takes on a different approach to the preceding implementations, as the compilers is written in Clojure. It is a hand-written recursive descent parser that requires a running Clojure environment in order to evaluate input source code, which is of the form of a subset of Clojure.

ClojureJS proposes the idea of Special Forms, which are JavaScript specific functionalities, as well as an informally-defined set of forms that is a subset of Clojure.

ClojureJS is perhaps the second best implementation of Clojure to JavaScript compilation, with its support of macros and a lot larger subset of Clojure than for instance LiScript. It is however, in my opinion, a non-extensible implementation of a trans-compiler and it does not provide any ease-of-use features, such as being able to generate an output .js file out of a given source input.

2.5 Professional and ethical issues

Throughout the report, I will make sure that I am not violating the British Computer Society Code of Conduct and Code of Good Practice and that I have applied their principles throughout the project.

This project, however, does not make any use of third party libraries and does not re-use any existing code base.

I will make all the effort to properly state and reference, whenever I am communicating someone else's ideas.

Chapter 3

Design & Specification

As previously described, the project aims to create an extensible Lispish to JavaScript compiler. In order to clearly define the possible constructs for writing programs in Lispish, we need to formalise our input language. As Lispish defines a subset of an existing language, it is therefore even more important to be clear on what is 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

Figure 3.1 illustrates the Lispish grammar. Grammar of the language formally defines the legal operators and operations that the language provides for writing programs in Lispish.

$$
\begin{array}{lcc}
F & ::= & (\text{let}\ \ [x\ F]\ \ (F)) \\
 & | & (\text{if}\ \ (F)\ \ F_1\ \ F_2) \\
 & | & (\text{defn}\ \ name\ \ [args*]\ \ (F)) \\
 & | & (\text{fn}\ \ [arg]\ \ (F)) \\
 & | & (\text{cond}\ \ (F_0)\ \ F_1\ \ (F_2)\ \ F_3) \\
 & | & (\text{s}\ \ args*) \\
 & | & (\text{recur}\ \ args*) \\
 & | & (\text{let}\ [X\ \ ((F_0))] \\
 & | & T \\
 & | & X
\end{array}
$$

where

$$
\begin{array}{lccl}
X & ::= & T & \\
T & ::= & () \mid N \mid B \mid s & * \text{ denotes multiple arity}
\end{array}
$$

operators

$$
\begin{array}{lcc}
N & ::= & n \mid (op\ N\ N) \\
B & ::= & b \mid (bop\ t1\ t2) \\
op & ::= & + \mid - \mid * \mid / \mid < \mid > \mid = \\
bop & ::= & and \mid or \mid not
\end{array}
$$

atomic

$$
\begin{array}{lcc}
s & ::= & String \\
n & ::= & Integer \\
b & ::= & true \mid false \\
() & ::= & List
\end{array}
$$

Figure 3.1: Lispish grammar

3.1.2 Evaluation relations (Big-Step Semantics)

Figure 3.2 describes the evaluation relations of Lispish. These relations therefore describe the constructs that can be used when writing Lispish programs. They, however, do not relate to the evaluation relations of the generated JavaScript code.

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 individually tested by unit tests.

Clojure allows developers to create programs using the REPL (Read Evaluation Print Loop), which is characteristic feature in modern 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.

3.3 Compiling Lispish to JavaScript

3.3.1 Compilation pipeline

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 the Clojure reader will prevent the code from being evaluated and it will pass it along down the pipeline to its respective emitters as illustrated on figure 3.3. 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 Clojure homoiconicity and using clojure.reader as a parser

Clojure being a homoiconic language, meaning that the code itself is described in terms of data structures that the language understands (s-expression forms, lists and in-built data structures), comes bundled with a Clojure reader[9] that can parse the text source file into objects of specific Clojure type. Those objects are essentially Clojure data structures that then are treated by the Clojure compiler and similarly in case of Lispish will be treated by the translator program, to generate corresponding JavaScript code.

As pointed out in the referenced [9] section of the Clojure documentation, the Clojure reader is represented by the `(reader )` function, that takes text as an input and produces the object represented by that text. This is also the entry point of our translator, as illustrated on 3.3 as the entry block in the flowchart. The translator will then generate different JavaScript constructs based on the type of the object that it comes across when recursively evaluating the type of each nested expression.

Opertors:

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

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

Atomic:

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

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

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

Forms (F):

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

$$\text{if true}\ \frac{t_0 \Downarrow \textsc{True} \qquad t_1 \Downarrow v}{(if\ (t_0)\ t_1\ t_2) \Downarrow v)}$$

$$\text{if false}\ \frac{t_0 \Downarrow \textsc{False} \qquad t2 \Downarrow v}{(if\ (t_0)\ t_1\ t_2) \Downarrow v)}$$

$$\text{cond}\ \frac{t_0 \Downarrow \textsc{False} \qquad t_1 \Downarrow \textsc{True} \qquad t_3 \Downarrow v}{(cond\ (t_0)\ t_2\ (t_1)\ t_3) \Downarrow v)}$$

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

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

$$\text{call to external function}\ \frac{args* \Downarrow v}{(s\ args*) \Downarrow v}$$

$$\text{recur}\ \frac{args* \Downarrow v}{(recur\ args*) \Downarrow v}$$

Recur emits all of its arguemnts and provides it as arguments to JavaScript `arguments.callee(args*)` function.
The `(s )` form that uses a string as a function, allows for invoking named function recursively, as well as any in-built JavaScript functions and library (e.g. jQuery) functions, as described in 4.3.

Figure 3.2: Lispish evaluation relations (Big-Step Semantics)

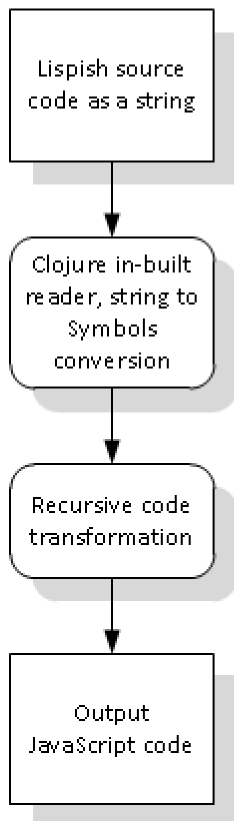

Figure 3.3: Abstract compilation of Lispish to JavaScript.

Chapter 4

Implementation and Testing

Following the formal definition of the Lispish language and briefly describing the operations of the compiler, we shall dive into the construction of it. Along the way, I will explain the concepts behind most of the mechanics of the compiler.

In the following sections, we will also test the implementation by means of examples of an actual compilation. At the end, we will look into automating the tests by means of using a Clojure testing API.

4.1 Building the compiler

This section will describe the operations of the compiler and the fundamental concepts behind how the compiler translates the input Lispish code to JavaScript.

Lispish translator is implemented as a single pass compiler. This design decision comes from the fact that the compiler has been implemented in Clojure, which is a strictly functional language and in order to sustain the immutability property throughout the compiler and not violate common idioms, the compiler avoids using state at all costs. It was therefore difficult to perform multi-pass compilation over the same code, as it is done in other commercial compilers. As a consequence of this, the entire implementation is built around recursively invoking a set a functions, which at the end fold to yield a JavaScript string as a result.

4.1.1 Abstract Structural Binding

Abstract Structural Binding allows for de-structuring any data structure to a corresponding argument in function parameters or a let form, creating locally scoped bindings. 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 binds the first two elements of the collection to x1 x2, in the above case, x1 will yield 1 and x2 null.

Lispish uses de-structuring for generating all of its forms. Take for instance the signature of a `(emit-defn)` function responsible for expanding and generating the equivalent JavaScript named function code:

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

In order to split the provided input source code `(defn ))` form into its respective elements, the performs a structural binding of the function arguments. The bindings are then used to generate the equivalent JavaScript code.

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 convenience placeholder for the head of the whole expression. The actual expression begins to bind from the [defn name [arg & more] & rest] arguments.

```
(defn emit-defn [type [defn name [arg & more] & rest]]
    (str "function " (if (= "~" name) "" name) "("

         ;; Output the argument names
         (if (nil? more) arg (str arg ", " (clojure.string/join ", " more)))
          ") {return "

         ;; Emit body of the function
         (emit rest)

         "}")))
```

Figure 4.1: Emit-defn function

Figure 4.1 illustrates the body of the `(emit-defn)` function and how the bindings acquired upon the function call are used to then generate the corresponding JavaScript quite. At first, we are checking if the `name` binding is a character, which is a special case meaning that the function is anonymous (we are passing it from the emit-fn with the rest of the expression, to decrease code count) and an empty string in place of the function name needs to be generated. The function then outputs the argument names. In case if there are multiple arguments, it will output an arguments string with the arguments separated by commas. At the end it emits the body `(emit rest)` of the function.

The optional `more` in the arguments list allows for an arbitrary length of the function arguments and the optional rest is for the expression that follows the named function.

4.1.2 Recursive Expansion

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

Figure 4.2 illustrates the flow chart of the compiler. It covers most of the operations of the compiler, except for the details on how multi-arity s-expressions are handled.

To illustrate how in practice the recursive expansion is performed, lets

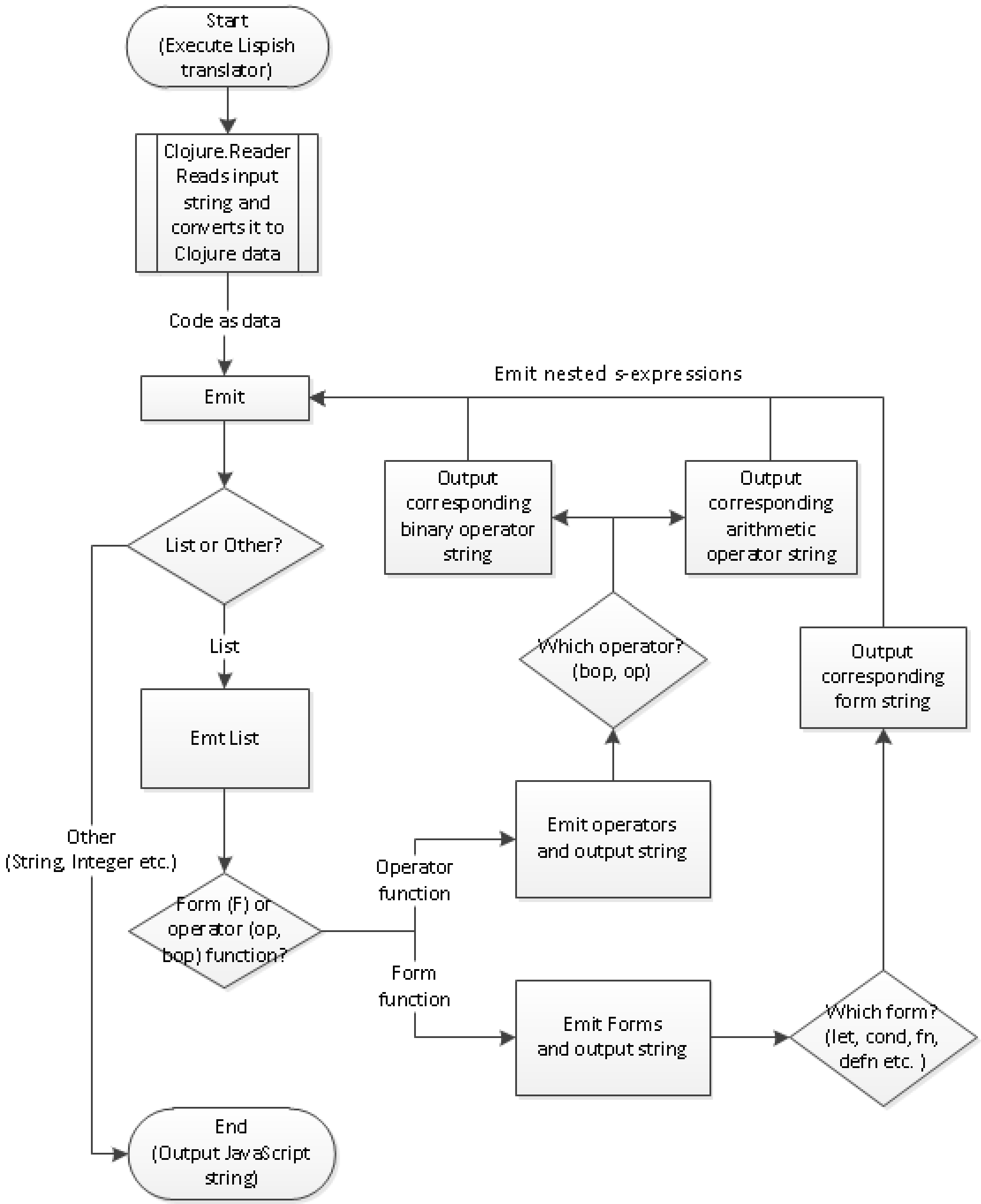

Figure 4.2: Flow chart of *Lispish to JavaScript* compilation.

consider how a single form gets expanded and also how its equivalent JavaScript code is generated.

```
(defn emit [expressions]
  "Take an s-expression and emit its corresponding JavaScript form"
  (do
    (println "Emit Lispish: " expressions)
    (cond
      (nil? expressions) "null"
      (symbol? expressions) (str expressions)
      (seq? expressions) (emit-list expressions)
      (integer? expressions) (str expressions)
      (float? expressions) (str expressions)
      (string? expressions) (str expressions)
      :else (str expressions))))
```

Figure 4.3: Top level function of the recursive expansion

Figure 4.3 is the top level function of the recursive translator. It is responsible for determining s-expression's type. This is possible as whenever the translator gets executed and a source file is provided as its input, the Clojure Reader is used to read the input source and as a result, it outputs the code as data, more precisely, as lists. These lists can be then checked for type, as the Clojure Reader is responsible for parsing and giving each symbol its corresponding type.

As illustrated on the 4.2 flowchart, there is indeed two cases for the emit function. The argument is either a list and it therefore needs to be expanded or it's one of the generic types, e.g. Integer or String and therefore needs to be outputted as a string already at this point.

4.1.3 Forms with multiple arity

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 compiler to a JavaScript string, map and reduce constructs have been used.

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

Reduce

Reduce is a function that takes a function, an optional value (or an s-expression) and a collection as an argument. It reduces or in other words folds a given collection (and an optional value) through the application of a function to a collection, to a single result.

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

that yields

```
"1123"
```

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 the list `("1" "2" "3")`.

To now put the map reduce constructs into perspective with Lispish, figure 4.4 illustrates how a multiple arity cond (allowing practically unbound list of tests) is implemented.

```
(defn emit-cond [head [name & rest]]
  (let [rev (reverse (partition 2 rest))]
    (reduce
     (fn [a b] (str "(" (emit (first b)) "?" (emit (second b)) ":" a ")"))
        (str (emit (second (first rev))))
        (drop 1 rev))))
```

Figure 4.4: emit-cond source code

Given an arbitrary number of `(test) result` tuples for the input `(cond )`, the `(emit-cond)` form first partitions the input into test and expression tuples, then reverses the tuples, so that the originally last one appears at the front, allowing us to perform a right reduce (right fold) and then binds it to a local `rev` variable. For example, if `(emit-cond )` is invoked with the following arguments:

```
(< 5 2) false (> 3 2) true :else false
```

the content of the locally scoped `rev` will be

```
((:else false) ((> 3 2) true) ((< 5 2) false))
```

The reduce function then applies the anonymous function to the first value, which is the result of `(str (emit (second (first rev))))`, which in this example happens to be the `false` symbol, as it is grabbed from the first tuple `(:else false)` as the second element. Reduce is then applied to the second, third etc. element of the collection, in this case the `((> 3 2) true) ((< 5 2) false)`, whilst the overall result is accumulated in `a`.

1.	`a: false`	`b: ((> 3 2) true)`
2.	`a: ((3>2)?true:false)`	`b: ((< 5 2) false)`
3.	`a: ((5<2)?false:((3>2)?true:false))`	b:

Figure 4.5: Reduction of a cond with multiple arguments

Figure 4.5 illustrates a table of how each reduction step is performed in terms of the two arguments of the function passed to reduce. Variable `a` accumulates the overall result, whilst `b` is the current element of the `(cond )` that is being converted to JavaScript ternary expression.

4.1.4 Implementing (recur) with JavaScript arguments.callee

One of the peculiar implementation decisions was implement the `(recur )` form in terms of an `arguments.callee` invocation from within the JavaScript. This implementation allows us to generate a JavaSript recursive invocation call whenever the `(recur )` form has been used. It is true, however, that the `arguments.callee` is disallowed in JavaScript strict mode and the ECMAScript specifications as it impacts how much the JavaScript compiler can optimise the JavaScript code. When `arguments.callee` is used, the compiler cannot perform tail-recursive optimisations to reduce the overheads of normal recursive calls.

At the same time there are certain pros when deciding to use `arguments.callee` in case of Lispish. Most importantly, invoking a recursive call to an anonymous function is not possible in JavaScript, unless the `arguments.callee` is used. When a function that needs to evaluate to a value needs to be recursively called, it needs to be named, however Lispish allows for a anonymous functions that can be invoked recursively.

To implement `recur` otherwise, the higher level s-expression function name would have to be passed to the function responsible for emitting the `(recur)` form, or a multi pass compilation, which would in-line the function name wherever a recur is present, would have to be performed. In both cases, the translator implementation would increase in its complexity.

Another argument for using the `arguments.callee` call, is that in the initial research conducted, it seems that in some older JavaScript versions, it was not possible to invoke a function by its name from within a ternary expression of the form `(3<x ? true : someFunction(x) )`.

Lispish allows for using either `(recur )` as a syntactic sugar for the straight `(someFunction )` function name invocation, therefore if the `arguments.callee` cannot be used due to performance reasons or wanting to comply with the ECMAScript standard for validations, a normal function invocation can be used.

4.2 Testing

The section above described the operations that are part of the compilation, but they did not provide any examples of an actual compilation. In this section we will take a look at some examples of how our Lispsh to JavaScript compiler works. To illustrate the compilation, I will demonstrate the output of the recursive expansion that the compiler performs on the given Lispish program string. Each line of the compilation trace will correspond to a level in the recursion.The recursive folding will be done implicitly, therefore it does not appear in the compilation traces.

The examples are invoked from the interactive REPL, but later in this section I will illustrate how Lispish translator can be used as a standalone Java JAR file, that takes Lispish (.lispish) source file as an input and produces an equivalent JavaScript code in another file (.js).

Let's begin our tests by a simple nested arithmetic expression, illustrated on figure 4.6.

```
lispish.core> (lisp-to-js "(+ 2 (* 3 4))")
(+ 2 (* 3 4))
Emit Lispish:  (+ 2 (* 3 4))
Emit-list head:  + , tail:  (2 (* 3 4))
Emit-op, head:  + , tail:  (2 (* 3 4))
Emit Lispish:  2
Emit Lispish:  (* 3 4)
Emit-list head:  * , tail:  (3 4)
Emit-op, head:  * , tail:  (3 4)
Emit Lispish:  3
Emit Lispish:  4
"(2+(3*4))"
```

Figure 4.6: Emit simple arithmetics example

As we can see from the recursive trace illustrated on figure 4.6, our recursion begins with passing the Lispish source code to the initial `(emit)` form, which then begins the recursion. At first, our s-expression is of the form `(+ 2 (* 3 4))`, which is a a whole is a list (an s-expression). 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 (* 3 4))` 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 (* 3 4))` which reaches the bottom of the recursion in one step for the first `(2)` and in multiple steps for the second `(* 3 4)` as it again invokes the same recursive steps and reduces the second list to a string. It then reduces the result of both to a final string concatenated with the two operators as follows `"(2+(3*4))`, producing an equivalent JavaScript code.

The same procedure is repeated for all of the `op`, as well as `bop` type of expressions.

In order to test the usability of the translator, we need to test it with more complex examples of programs that could be written in Lispish, given its grammar and the forms that it supports.

```
(defn is_prime [num]
  (let [prime_over_two
         (fn [num factor]
           (if (> factor (Math.sqrt num))
               true
               (if (= 0 (mod num factor))
                   false
                   (recur num (+ 2 factor)))))]
    (cond
      (< num 2) false
      (= 2 num) true
      (= 0 (mod num 2)) false
       :else (prime_over_two num 3))))
```

Figure 4.7: Lispish naive primality checking source file

Program listed in figure 4.7 is an implementation of a naive primality checking written in Lispish.

Provided the JavaScript output listed in 4.8, the code should be parsable

```
function is_prime(num) {return (function(prime_over_two) {
return ((num<2)?false:((2==num)?true:((0==(num%2))?false:
prime_over_two(num, 3)))) })(function (num, factor) {
return ((factor>Math.sqrt((num)) ? (true):
(((0==(num%factor)) ? (false):(arguments.callee(num, (2+factor))))))})}
```

Figure 4.8: Naive primality testing Lispish to JavaScript output

by a JavaScript interpreter. To test this, we will use Google Chrome JavaScript console.

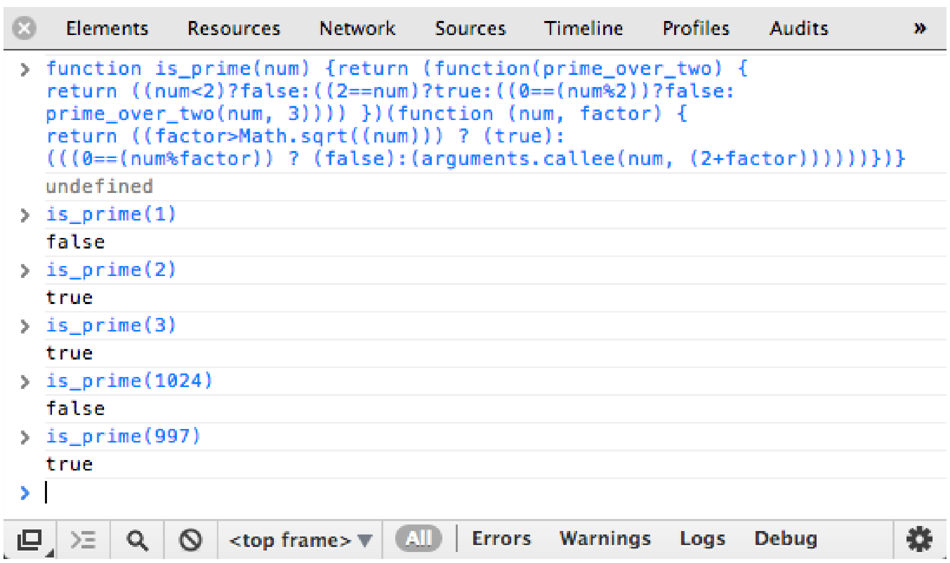

Figure 4.9: Compiled Lispish program for naive primality checking, being tested in Google Chrome JavaScript console and yielding right results.

Figure 4.9 illustrates testing of the naive primality test program as it is fed into the Google Chrome's JavaScript console. At the moment of pasting the translator's output to the console, the console yields `undefined`, meaning that a function has been successfully parsed and defined as `is_prime(num)`.

The `is_prime` function is then tested for the first three natural numbers: 1, 2, 3. 1 evaluate to false, whereas the first two prime numbers, 2 and 3 evaluate to true. We then test the prime 997, which also correctly evaluates to true and 1024 which correctly evaluates to false. As this paper does not intend to prove the correctness of the results of the JavaScript programs that the users of the translator writes, we can therefore conclude this test by saying that the translator produced correct JavaScript code.

4.2.1 Deploying and Using Lispish

The end goal of this project was to be able to compile a source Lispish program to an equivalent JavaScript program. It is, however, not ideal to have to perform compilation in an interactive REPL, where Clojure environment is set up.

To solve this problem, the Lispish compiler is compiled as a standalone JAR file that can be executed in any environment equipped with the Java Runtime Environment. This is possible as the JAR file bundles the Clojure language itself, as well as all of its dependencies and our Lispish compiler. It exposes the application through a simple static main method, which serves as an entry point to programs execution, similarly to standard Java applications.

There are three simple ways to compile a Lispish program to JavaScript.

The first method is to execute the Lispish jar file and provide simple source code as a command line argument.

Code as argument

```
bash$ java -jar lispish-1.0.jar "(+ 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)
```

Given as an input a prefix s-expression of (+ 2 2), the program yields an expected result, which is an equivalent in-fix (2+2).

This approach is fine for trivial examples that do not span across multiple lines, it is however not optimal when we want to compile a Lispish program file to an equivalent JS. It, therefore, only supports compiling programs made of a single s-expression like in the example above, meaning it will not compile two `defn` forms or two separate s-expressions that are not part of the same list.

Source (Lispish) and destination (JavaScript) file as argument

In order to compile a Lispish source file to an equivalent JavaScript source file, our compiler accepts two command line options:

```
["-in" "--input" "REQUIRED: Path to Lispish source code."]
["-out" "--output" "OPTIONAL: Path to JavaScript output file."]
```

`-in` or equivalently `--input`, that should follow with a path to a Lispish source file, as well as an optional `-out` or equivalently `--output`, that should follow with the name of the output source file.

To demonstrate how compilation of one source file to another is performed, figure 4.7 illustrates the content of the `test.lispish` file containing the same example of naive primality checking program written in Lispish, used in the implementation section.

We can then execute the compiler passing in the `--input` and `--output` arguments, as follows.

```
bash$ java -jar lispish-1.0.jar --input test.lispish --out test.js
```

The `--input` argument specifies path to the `test.lispish` file and `--output` argument is the path of the file to be generated. The compiler will print out all of the computation steps to the console, but the final result, that is, the JavaScript output, will be written to the file specified in the `--output` path.

Figure 4.8 illustrates the content of the generated `test.js` file with the JavaScript equivalent of the previously shown Lispish, naive primality testing, program.

I have purposely omitted the recursive trace of the compilation, as the recursion is invoked multiple times and it produces a very long output. The recursive trace is attached to the appendix A.1. It gives a very good idea how a more complex program gets translated by outputting each one of the forms as a string, also providing name of the function responsible for translating it.

As we can see, the Google Chrome web browsers console can evaluate the function and when executed with parameters, it yields the right result.

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[16] 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) {return
         ((n<2) ? (1):((n*factorial(((n-1))))))}"
        (lisp-to-js
          "(defn factorial [n]
             (if (< n 2) 1 (* n (factorial (- n 1)))))"))))
```

The above snippet is just one out of many tests included in appendix A.2.

4.2.3 Writing Lispish programs that interact with JavaScript functions

As a consequence of Lispish design, the language allows for forms that begin with an arbitrary string as a function name. During the translation, the expression with an arbitrary string as a function name is translated to a JavaScript function call with the same name. This allows the user of the language to use JavaScript built-in function names in their Lispish source code. An example of this has been illustrated in the example show on figure 4.7, where at some point of the computation, the Math.sqrt has been invoked.

```
(defn emit-call [head [name args & rest]]
  (println "Emit-call, name: " name ", args: " args ", rest: " rest)
  (str name "("
       (if (nil? rest)
         (str "(" (emit args) ")")
         (str (str (emit args)) ", "
          (clojure.string/join ", " (map emit rest)))) ")"))
```

Figure 4.10: emit-call source code

Figure 4.4 illustrates the `emit-call` function that is responsible for generating JavaScript code for recursive calls, as well as functions with arbitrary names, used for interacting with the browser and in-built JavaScript functions. As with the function responsible for generating code for JavaScript functions, it concatenates the optional function name, used here for invoking in-built JavaScript functions, with the emitted arguments to that function. In case if a Lispish form begins with a recursive call, the JavaScript `arguments.callee` is passed as a function name to the above `emit-call`. When generated, it outputs JS of the form of `arguments.callee(x, y, z)`, which tells the JS interpreter to recursively invoke the function with arguments `x, y, z`.

At first the emit-call might seem like a flaw, but it's a powerful feature, as it allows for invoking JavaScript functions responsible for interacting with the browser.

4.3 Writing Lispish programs using jQuery functions to interact with the browser's Document Object Model (DOM)

The in-built JavaScript functions that provide an interface for manipulating content of the Document Object Model are inherently imperative. For example, modifying a content of a `<div id=c̈ontenẗ>test</div>` element requires us to use an assignment operator that will modify the state of the DOM.

```
document.getElementById("test").innerHTML = "some text";
```

The JavaScript that is generated out of Lispish does not allow for imperative assignments, as an assignment is done by means of an an argument passed to an anonymous function, which binds to the functions argument, as in the following example:

```
(function(element) { element.innerHTML("some text") })(".test")
```

This code would fail in JavaScript, as the `.innerHTML` is an object's property. Lispish being a non object-oriented, functional language, does not provide this functionality and therefore for it to work, `innerHTML` would have to be a function.

As a coincidence, there exist JavaScript libraries, that are in fact very popular and provide wrappers over the standard JavaScript functions. An example of such library is a very popular jQuery[3]. One of the many functionalities that jQuery provides, is the possibility to modify content of a DOM element by passing the new content to a function. Similarly to the standard JS `.innerHTML` property, we can use jQuery's `.html()` function, that takes the new content as an argument and performs the DOM update internally.

```
(function(x) { return x.html(("test")) })($((".someDiv")))
```

The above code snippet has been generated from the following Lispish code:

```
(let [x ($(".someDiv"))] (x.html "test"))
```

The JavaScript functions takes a jQuery `$(".someDiv")` DOM node object and passes it as an argument to the preceding anonymous function. It then invokes the `.html()` function with the text `test` as an argument, which replaces the content of the DIV with the text.

Chapter 5

Evaluation

The implementation of Lispish to JavaScript was supposed to serve as an example of how a programming language translator can be implemented using a functional language.

Due to its functional nature and immutability, it does not suffer from errors caused by inconsistent state, as there is no state. The compiler will always produce a result. The result however is not guaranteed to be correct, if an incorrect input has been provided.

The parsing of the language is greatly simplified and inherently free of error, as the translator takes an advantage of the inherent feature of the Clojure language, which is its homoiconity and the Clojure reader that is responsible for parsing the input source files and producing the corresponding objects, as described in the design section of the report.

Appendix A.2 provides a set of unit tests that include multiple real applications of our translator by translating small Lispish programs including Fibonacci sequences, factorial, ackermann and naive primality checking to its equivalent JavaScript programs, as well as smaller tests that check the correctness of single forms. The unit tests included in the above mentioned appendix all successfully pass. The compiler also successfully compiles a given input file to an output file with name of our choice.

As a result of design decisions, Lispish is not strictly a subset of Clojure, as there is a small number of cases, where certain form inputs are not allowed. These exceptions include:

Lispish/Clojure inconsistencies:

- The `(let )` form only supports 1 argument, but it can be anything, including an anonymous function just like in strict Clojure.
- Strict Clojure allows for `(cond )` with an optional :else argument at the end. Due to a design decision, `:else` is strictly required.

The JavaScript code that is generated by the translator does not provide any optimisations and it makes a heavy use of recursion, which is of course not optimal for any real applications. It is not certain how severely this implementation affects the performance of the translated code, as the JavaScript V8 compiler could, in theory, optimise all of the `arguments.callee` recursive calls using tail recursion. This paper does not provide any benchmarks as to how the generated code performs compared to code for programs that would be hand written in JavaScript.

The grammar of Lispish allows for identifier names separated with the `-` dash symbol, which after translation, results in an invalid JavaScriot, as the dash character cannot be used as part of an identifier in JavaScript.

5.1 Missing parts

5.1.1 Error handling

The compiler does not provide any facility for error reporting during the compilation.

The compiler does not have any means of validating the JavaScript code. This could be incorporated by means of bundling a JavaScript validator that could simply analyse the code before it's served to an output file. This was however not part of the initial design and due to time constraints has not been implemented.

The biggest issue with the compiler is that it does not actually parse the input string before the compilation is performed. This caveat removes the possibility of determining if the input source, that is Lispish, is actually valid. Providing an invalid Lispish source code would still result in a JavaScript output, but the generated code would be malformed and would not execute in a browser. This is both true for semantical, as well as syntactical errors.

5.2 Evaluationg of Lispish against other Lisp to JavaScript implementations

Lispish does not try to compete with other compilers such as ClojureScript, as they are community-driven, mature and production ready that not only provide a much larger language support, but also the very crucial optimisations that a toy implementation such as Lispish does not provide.

In its current version, ClojureScript does not define a subset of Clojure that it can compile, therefore it is understandable that it provides a full language coverage. Lispish in retrospect defines limited language constructs that can be used to build simple programs and therefore, it is nowhere as powerful as ClojureScript.

In contrast with Outlet, Lispish takes an advantage of a popular Clojure language and proposes to translate a small subset of the language to JavaScript. Outlet on the other hand defines it's own original Lisp-like syntax that must be understood by a Clojure or any other Lisp programmer, before even a trivial program can be implemented. One of the advantages of Outlet is that it allows the programmer to extend the language using macro functions, whereas Lispish does not provide macros and thus has a much lower dynamism compared to Outlet.

Chapter 6

Conclusions and Future Work

Functional compilers provide an elegant alternative to compilers written in imperative languages, it is however not trivial to implement one, given the nature of functional programs.

The translator at its current state could be a good foundation for initiating a collaboration with the open source community, that could have interest in extending it.

It was not in the scope of this paper to provide any formal proofs of the correctness of the translations. This, however, would be a very achievable target, mainly due the functional properties of the compiler, namely the one-input/one-output property. Providing formal proves of the correctness of the translator and even further, of the generated translations could be a good project for Master Thesis and beyond.

6.1 Future work

6.1.1 Macros

It is typical for Lisp languages to provide a way to define new constructs in terms of already existing language constructs. For this to happen, a language needs to support macros. Macros provide a way to extend the language at compile time. Due to time constraints, I was unable to perform sufficient research into how provide the flexibility of macros, yet still being able to parse the code correctly.

6.1.2 Parser

In order for Lispish translator to be a true compiler, it would need a parser that can decide whether the input string, that is the Lispish program, is in fact a correct one. As mentioned in the section above, it does not provide any error detection facility and this therefore would be the first step for proper error handling.

6.1.3 JavaScript validator

For Lispish to be truly useful, its translator would need have to have a JavaScript validation in place. The parser used for validation could flag, for example, at which point there is an error in the generated JavaScript code and therefore simply the task of debugging the translated code.

JavaScript that is not syntactically correct due to the faulty input Lispish is only going to decrease the productivity of the developer, which goes against the core idea of building an abstraction over an imperative language.

References

[1] Javascript v8 vs python 3.0 programs performance benchmark, . URL http://benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=v8&lang2=python3&data=u32. Accessed 22 April 2013.

[2] Javascript v8 vs ruby 2.0 programs performance benchmark, . URL http://benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=v8&lang2=yarv&data=u32. Accessed 22 April 2013.

[3] jquery. URL http://jquery.com/. Accessed 22 April 2013.

[4] Javascript v8 vs python 3.0 programs performance benchmark. URL http://www.windowsazure.com/en-us/develop/nodejs/. Accessed 22 April 2013.

[5] A retrospective on paradigms of ai programming. URL http://norvig.com/Lisp-retro.html. Accessed 23 April 2013.

[6] Liscript, 2013. URL https://github.com/viclib/LiScript. Accessed 17 April 2013.

[7] Oleg Andreev. Recursive descent parser in javascript, 2013. URL http://blog.oleganza.com/post/106246432/recursive-descent-parser-in-javascript. Accessed 17 April 2013.

[8] Rich Hickey. Clojure, 2008. URL http://clojure.org/. Accessed 17 April 2013.

[9] Rich Hickey. Clojure reader, 2008. URL `http://clojure.org/reader`. Accessed 17 April 2013.

[10] Rich Hickey. clojure-clr, 2009. URL `https://github.com/clojure/clojure-clr`. Accessed 17 April 2013.

[11] Rich Hickey. Clojurescript, 2011. URL `https://github.com/clojure/clojurescript`. Accessed 17 April 2013.

[12] Ram Krishnan. A naive clojure to javascript translator, 2011. URL `https://github.com/kriyative/clojurejs`. Accessed 17 April 2013.

[13] James Long. Outlet, 2012. URL `https://github.com/jlongster`. Accessed 17 April 2013.

[14] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, 1991. ISBN 1558601910. URL `http://www.amazon.co.uk/Paradigms-Artificial-Intelligence-Programming-Studies/dp/1558601910/ref=sr_1_1?ie=UTF8&qid=1366716338&sr=8-1&keywords=paradigms+of+artificial+intelligence+programming`.

[15] Ronald L. Rivest. S-expressions memo, mit, 1997. URL `http://people.csail.mit.edu/rivest/Sexp.txt`. Accessed 25 April 2013.

[16] Stuart Sierra. A unit testing framework - api for clojure.test., 2011. URL `http://richhickey.github.io/clojure/clojure.test-api.html`. Accessed 17 April 2013.

[17] Wikipedia. S-expressions. URL `http://en.wikipedia.org/wiki/S-expression`. Accessed 25 April 2013.

Appendix A

Extra Information

A.1 Compilation trace of Lispish naive primality testing to JavaScript

```
Emit Lispish:  (defn is_prime [num] (let [prime_over_two (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))))
Emit-list head:  defn , tail:  (is_prime [num] (let [prime_over_two
(fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false :else
(prime_over_two num 3))))
Emit-forms, head:  defn , full expression:  (defn is_prime [num]
(let [prime_over_two (fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false :else
(prime_over_two num 3))))
Emit-defn, name:  , arg:  num , arg tail:  nil , rest:  ((let [prime_over_two
(fn [num factor] (if (> factor (Math.sqrt num)) true
```

```
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false :else
(prime_over_two num 3))))
Emit Lispish:  ((let [prime_over_two (fn [num factor] (if (> factor
(Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))))
Emit Lispish:  (let [prime_over_two (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3)))
Emit-list head:  let , tail:  ([prime_over_two (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3)))
Emit-forms, head:  let , full expression:  (let [prime_over_two
(fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false
:else (prime_over_two num 3)))
type:  let , let:  let , x:  prime_over_two , y:  (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor))))) , body:  (cond (< num 2) false
(= 2 num) true (= 0 (mod num 2)) false :else (prime_over_two num 3))
Emit Lispish:  prime_over_two
Emit Lispish:  (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))
Emit-list head:  cond , tail:  ((< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))
Emit-forms, head:  cond , full expression:  (cond (< num 2) false
(= 2 num) true (= 0 (mod num 2)) false :else (prime_over_two num 3))
```

```
Emit-cond, head:  cond , name:  cond , rest:  ((< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3)) , reverse after
partitioning:  ((:else (prime_over_two num 3)) ((= 0 (mod num 2)) false)
((= 2 num) true) ((< num 2) false))
Emit Lispish:  (prime_over_two num 3)
Emit-list head:  prime_over_two , tail:  (num 3)
Emit-forms, head:  prime_over_two ,
full expression:  (prime_over_two num 3)
Emit-call, name:  prime_over_two , args:  num , rest:  (3)
Emit Lispish:  num
Emit Lispish:  3
Emit Lispish:  (= 0 (mod num 2))
Emit-list head:  = , tail:  (0 (mod num 2))
Emit-op, head:  = , tail:  (0 (mod num 2))
Emit Lispish:  0
Emit Lispish:  (mod num 2)
Emit-list head:  mod , tail:  (num 2)
Emit-op, head:  mod , tail:  (num 2)
Emit Lispish:  num
Emit Lispish:  2
Emit Lispish:  false
a:  ((0==(num%2))?false:prime_over_two(num, 3)) , b:  ((= 2 num) true)
Emit Lispish:  (= 2 num)
Emit-list head:  = , tail:  (2 num)
Emit-op, head:  = , tail:  (2 num)
Emit Lispish:  2
Emit Lispish:  num
Emit Lispish:  true
a:  ((2==num)?true:((0==(num%2))?false:prime_over_two(num, 3))) , b:
((< num 2) false)
Emit Lispish:  (< num 2)
```

```
Emit-list head:  < , tail:  (num 2)
Emit-op, head:  < , tail:  (num 2)
Emit Lispish:  num
Emit Lispish:  2
Emit Lispish:  false
Emit Lispish:  (fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))
Emit-list head:  fn , tail:  ([num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))
Emit-forms, head:  fn , full expression:  (fn [num factor] (if (> factor
(Math.sqrt num)) true (if (= 0 (mod num factor)) false (recur num
(+ 2 factor)))))
Emit-defn, name:  , arg:  num , arg tail:  (factor) , rest:  ((if (> factor
(Math.sqrt num)) true (if (= 0 (mod num factor)) false (recur num
(+ 2 factor)))))
Emit Lispish:  ((if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))
Emit Lispish:  (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit-list head:  if , tail:  ((> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit-forms, head:  if , full expression:  (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit-if, condition:  (> factor (Math.sqrt num)) , true-form:  true ,
false-form:  ((if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit Lispish:  (> factor (Math.sqrt num))
Emit-list head:  > , tail:  (factor (Math.sqrt num))
Emit-op, head:  > , tail:  (factor (Math.sqrt num))
Emit Lispish:  factor
Emit Lispish:  (Math.sqrt num)
Emit-list head:  Math.sqrt , tail:  (num)
```

```
Emit-forms, head:  Math.sqrt , full expression:  (Math.sqrt num)

Emit-call, name:  Math.sqrt , args:  num , rest:  nil

Emit Lispish:  num

Emit Lispish:  true

Emit Lispish:  ((if (= 0 (mod num factor)) false (recur num (+ 2 factor))))

Emit Lispish:  (if (= 0 (mod num factor)) false (recur num (+ 2 factor)))

Emit-list head:  if , tail:  ((= 0 (mod num factor)) false (recur num
(+ 2 factor)))

Emit-forms, head:  if , full expression:  (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))

Emit-if, condition:  (= 0 (mod num factor)) , true-form:  false , false-form:
((recur num (+ 2 factor)))

Emit Lispish:  (= 0 (mod num factor))

Emit-list head:  = , tail:  (0 (mod num factor))

Emit-op, head:  = , tail:  (0 (mod num factor))

Emit Lispish:  0

Emit Lispish:  (mod num factor)

Emit-list head:  mod , tail:  (num factor)

Emit-op, head:  mod , tail:  (num factor)

Emit Lispish:  num

Emit Lispish:  factor

Emit Lispish:  false

Emit Lispish:  ((recur num (+ 2 factor)))

Emit Lispish:  (recur num (+ 2 factor))

Emit-list head:  recur , tail:  (num (+ 2 factor))

Emit-forms, head:  recur , full expression:  (recur num (+ 2 factor))

Emit recur, head:  recur , expression:  (recur num (+ 2 factor))

Emit-call, name:  arguments.callee , args:  num , rest:  ((+ 2 factor))

Emit Lispish:  num

Emit Lispish:  (+ 2 factor)

Emit-list head:  + , tail:  (2 factor)
```

```
Emit-op, head:  + , tail:  (2 factor)
Emit Lispish:  2
Emit Lispish:  factor

function is_prime(num) {return (function(prime_over_two) { return ((num<2)
?false:((2==num)?true:((0==(num%2))?false:prime_over_two(num, 3)))) })(
function (num, factor) {return ((factor>Math.sqrt((num))) ? (true):(((0==(
num%factor)) ? (false):(arguments.callee(num, (2+factor))))))})}
```

A.2 Test coverage of the naive Clojure recursive-descent-parser implementation

```
(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 logical-or
  (is (= "((5>10)||(10>5))" (lisp-to-js "(or (> 5 10) (> 10 5))"))))
(deftest logical-and
  (is (= "((5>10)&&(10>5))" (lisp-to-js "(and (> 5 10) (> 10 5))"))))
(deftest logical-and
  (is (= "(!(5>10))" (lisp-to-js "(not (> 5 10))"))))
(deftest if-form
```

```
  (is (= "((5>10) ? (true):(false))" (lisp-to-js "(if (> 5 10)\"true\" \"false\")"))))
(deftest fn-form
  (is (= "function (x) {return (x*x)}" (lisp-to-js "(fn [x] (* x x))"))))
(deftest let-form
  (is (=
  "(function(x) { return (x*x) })(2)"
  (lisp-to-js "(let [x 2] (* x x))"))))
(deftest let-lambda-function
  (is (=
  "(function(times-five) { return times-five((5)) })(function (x) {return
  (x*5)})"
  (lisp-to-js "(let [times-five (fn [x] (* x 5))] (times-five 5))"))))
(deftest defn-form
  (is (= "function square(x) {return (x*x)}"
  (lisp-to-js "(defn square [x] (* x x))"))))
(deftest fibonacci-example
  (is (= "function fib(n) {return ((n<2) ? (1):((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) {return ((n<2) ? (1):((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) {return
  ((m==0)?(n+1):((n==0)?ackermann((m-1), 1):
  ackermann((m-1), ackermann(m, (n-1)))))}"
  (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)))))"))))
(deftest primality-checking-program
  (is (=
  "function is_prime(num) {return (function(prime_over_two) { return ((num<2)
  ?false:((2==num)?true:((0==(num%2))?false:prime_over_two(num, 3)))) })(
  function (num, factor) {return ((factor>Math.sqrt((num))) ? (true):(((0==(
  num%factor)) ? (false):(arguments.callee(num, (2+factor))))))})}"
  (lisp-to-js
  "(defn is_prime [num]
    (let [prime_over_two
            (fn [num factor]
              (if (> factor (Math.sqrt num))
                  true
                  (if (= 0 (mod num factor))
                      false
                      (recur num (+ 2 factor)))))]
       (cond
         (< num 2) false
         (= 2 num) true
         (= 0 (mod num 2)) false
         :else (prime_over_two num 3))))"))))
```

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

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary. – Daniel Zurawski, 17 April 2013.

C.1.1 lispish/project.clj

```
(defproject ClojureToJavaScript "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.3.0"]
                 [org.clojure/tools.trace "0.7.3"]
                 [org.clojure/tools.cli "0.2.1"]]
  :main lispish.core)
```

C.1.2 lispish/src/lispish/core.clj

```
(ns #^{:author "Daniel Zurawski"
       :doc "A simple Lisp to JavaScript transcompiler written in Clojure."}
  lispish.core
  [:require
   [clojure.string :as str]]
```

```
  [:use
   [clojure.walk]
   [clojure.tools.trace]
   [clojure.tools.cli :only (cli)]]
  (:gen-class :main true))


(def op (set ['mod '+ '- '* '/ '> '< '=]))
(def bop (set ['or 'and 'not]))
(def forms (set ['recur 'let 'if 'fn 'defn 'cond]))

;; Clojure is a single pass compiler, thus we have to use forward declaration
;; if we need to use a function before it's declared
(declare emit-list)

(defn emit [expressions]
  "Take an s-expression and emit its corresponding JavaScript form"
  (do
    (println "Top level - Emit Lispish: " expressions)
    (cond
      (nil? expressions) "null"
      (symbol? expressions) (str expressions)
      (seq? expressions) (emit-list expressions)
      (integer? expressions) (str expressions)
      (float? expressions) (str expressions)
      (string? expressions) (str \" expressions \")
      :else (str expressions))))

;; Abstract Structural Binding - + falls in type, + in op and 2 2 in tail
(defn emit-op [type [op & tail]]
  "Emit s-expression with single operators and two arguments"
```

```
  (do (println "Emit-op, head: " op ", tail: " tail)
      ;; Interlace the arguments with the operator
      (if (= op 'not)
        (str "(!" (emit tail) ")")
        (str "(" (clojure.string/join
                (str (cond (= op '=) "=="
                           (= op 'mod) "%"
                           (= op 'or) "||"
                           (= op 'and) "&&"
                           :else op))
                (map emit tail))
           ")"))))

(defn emit-let [type [let [x y] body]]
  (println "Emit-let, x: " x ", y: " y ", body: " body)
        (str "(function(" (emit x) ") { return "  (emit body)  " })(" (emit y)
        ")" ))

(defn emit-if [type [if condition true-form & false-form]]
  (println "Emit-if, condition: " condition ", true-form: " true-form ", false-
  form: " false-form)
  (str "("
       (emit condition)
       " ? ("
       (emit true-form)
       "):("
       (emit false-form)
       "))"))

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

```
    (println "Emit-defn, name: " ", arg: " arg ", arg tail: " more ", rest: " rest))
  (str (str "function " (if (= "~" name) "" name) "("
            (if (nil? more) arg (str arg ", " (clojure.string/join ", " more)))
             ") {return "
     (emit rest)
     "}")))

(defn emit-fn [head expression]
  (emit-defn head (concat (take 1 expression) '("~") (drop 1 expression))))

(defn emit-call [head [name args & rest]]
  (println "Emit-call, name: " name ", args: " args ", rest: " rest)
  (str name "("
       (if (nil? rest)
         (str "(" (emit args) ")")
         (str (str (emit args)) ", " (clojure.string/join ", " (map emit rest))
         )) ")"))

(defn emit-recur [head expression]
  (println "Emit recur, head: " head ", expression: " expression)
  (emit-call head (concat '("arguments.callee") (drop 1 expression))))

(defn emit-cond [head [name & rest]]
  (let [rev (reverse (partition 2 rest))]
    (println "Emit-cond, head: " head ", name: " name ", rest: " rest
    ", reverse after partitioning: " rev)
    (reduce
          (fn [a b] (do (println "a: " a ", b: " b ) (str "(" (emit (first b))
          "?" (emit (second b)) ":" a ")")) )
          (str (emit (second (first rev))))
          (drop 1 rev))))
```

```
(defn emit-forms [head expression]
  (do (println "Emit-forms, head: " head ", full expression: " expression)
      (cond (= head 'let) (emit-let head expression)
            (= head 'if) (emit-if head expression)
            (= head 'fn) (emit-fn head expression)
            (= head 'defn) (emit-defn head expression)
            (= head 'cond) (emit-cond head expression)
            (= head 'recur) (emit-recur head expression)
            :else (emit-call head expression) )))

(defn emit-list [expressions]
  (do
      (if (symbol? (first expressions))

        (let [head (symbol (first expressions))
              expressions (conj
                           (rest expressions) head)]

          (println "Emit-list head: " head
                   ", tail: " (rest expressions))
          (cond
            (or (contains? op head) (contains? bop head)) (emit-op head expressions)
            (contains? forms head) (emit-forms head expressions)

            :else (emit-forms head expressions)
            ))
        ;; Not safe, may run into stack overflow if this will be a list or not-
        recognized
        (emit (first expressions)))))
```

```
(defn lisp-to-js [forms]
  (let [code (read-string forms)]
    (println code)
    (emit code)))

(defn read-file-emit [st file-out]
    (let [form (read st false "")]
      (if (not (= form ""))
        (do
          (spit file-out (str (emit form) "\n")  :append true)
          (read-file-emit st file-out)))))

(defn read-file [file-in file-out]
  (with-open [r (java.io.PushbackReader.
                 (clojure.java.io/reader file-in))]
    (binding [*read-eval* false]
      (spit file-out "" :append false)
      (read-file-emit r file-out))))

(defn run
  "Print out the options and the arguments"
  [opts args]
  (cond
    (:input opts) (if (:output opts)
                    (read-file (:input opts) (:output opts))
                    (println "Please provide --output or -out, path where the output JavaSc
    (seq args)    (println (lisp-to-js (first args)))
    :else         (println "No path to input source code specified and no code given as arg

(defn -main [& args]
  (let [[opts args banner]
```

```
    (cli args
         ["-h" "--help" "Show help" :flag true :default false]
         ["-in" "--input" "REQUIRED: Path to Lispish source code."]
         ["-out" "--output" "OPTIONAL: Path to JavaScript output file."]
         )]
(when (:help opts)
  (println banner)
  (System/exit 0))
(if (or (:input opts) (seq args))
  (run opts args)
  (println banner))))
```