KING'S College LONDON

University of London

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 extendible, 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 present ways 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.

Daniel Marian Zurawski

16th November 2012

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

1 Introduction **3**
- 1.1 Report Structure 4

2 Background **5**
- 2.1 Functional programming 5
- 2.2 Lisp 6
- 2.3 JavaScript as a target language for the Lispish language 8
- 2.4 Existing Lisp to JS compilers 10

3 Report Body **12**
- 3.1 Section Heading 12

4 Design & Specification **13**
- 4.1 The Lispish language 13

5 Implementation and Testing **14**
- 5.1 Section Heading 14

6 Professional Issues **15**
- 6.1 Section Heading 15

7 Results/Evaluation **16**
- 7.1 Section Heading 16

8 Conclusion and Future Work **17**

A Extra Information **18**
- A.1 Test coverage of the naive Clojure recursive-descent-parser implementation 18
- Bibliography 18

B User Guide **21**
- B.1 Instructions 21

C Source Code **22**
C.1 Instructions 22

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" ....

With the uprise of high performance JavaScript compilers such as the Google V8 JavaScript Engine, raised the interest in creating programming language translator (compilers or in other words trans-compilers) from a different source programming language to JavaScript, thus making it portable as JavaScript can be run in any modern web browser. On the other hand, the Lisp family of programming languages offers a simplistic and elegant, yet powerful syntax based on parenthesised lists and expressions written in the prefix notation (Polish notation). I will investigate how such translation is performed and implement a simple compiler that can translate a subset of Lisp to JavaScript, using a Clojure programming language, which is a modern dialect of Lisp running on the Java Virtual Machine (JVM).

The wide adoption of the web over the recent years and the increasing interest in building Rich Internet Applications (RIA) resulted in big companies offer-

ing Software as a Service solution in the form of web applications, notably application from the Google Apps family, including Gmail, Google Calendar and Talk. To support development of such applications raised the need to support different programming paradigms and therefore compilers or in other words trans-compilers that could convert a source language (e.g. Java) into an optimised JavaScript code, here notably GWT (Google Web Toolkit). It was necessary to improve the efficiency of JavaScript interpreters and as a result tools such as the Google V8 JavaScript Engine [reference here] that compiles a platform independent JavaScript directly to machine code vastly improving the attractiveness of JavaScript. Shortly after that communities of different programming languages started to develop similar compilers from different source languages that could compile to the same JavaScript in order to make their code portable. 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) 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 and JavaScript and how both can be used to solve complex problems in Computer Science. The end result of this project is going to be a simple Lisp to JavaScript compiler, that given an input string (Lisp program) will be able to produce a plug and play executable JavaScript, that can be later executed. [**?**].

1.1 Report Structure

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 Functional programming

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). To make this abstraction possible, pure functional languages try to avoid using state and mutability, by ensuring that side effects that could introduce state changes are not possible thus guaranteeing referential transparency.

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/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 and 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.

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 we want to override its content.

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 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 called 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.

2.2.1 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 build a Syntax Tree (ST) or an Anotated Syntax Tree (AST). In our case, our input s-expressions can be treated as a ST on its own and thanks to the in-built functions, we can greatly simplify the compiler by making it less error prone.

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

```
// 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 descripting 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:

```
(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.

2.3.2 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.

2.3.3 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]

2.4 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 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

Report Body

The central part of the report usually consists of three or four chapters detailing the technical work undertaken during the project. **The structure of these chapters is highly project dependent**. They can reflect the chronological development of the project, e.g. design, implementation, experimentation, optimisation, evaluation, etc (although this is not always the best approach). However you choose to structure this part of the report, you should make it clear how you arrived at your chosen approach in preference to other alternatives. In terms of the software that you produce, you should describe and justify the design of your programs at some high level, e.g. using OMT, Z, VDL, etc., and you should document any interesting problems with, or features of, your implementation. Integration and testing are also important to discuss in some cases. You may include fragments of your source code in the main body of the report to illustrate points; the full source code is included in an appendix to your written report.

3.1 Section Heading

3.1.1 Subsection Heading

Chapter 4

Design & Specification

4.1 The Lispish language

4.1.1 Grammar rules

let: (let [binding] (expression))

if: (if (test-form) true-form false-form)

defn: (defn name [args*] (expression))

fn: (fn [arg] (expression))

cond: (cond (test-form) true-form (test-form2) true-form2)

4.1.2 Inference rules

$$\text{let}\ \frac{expr \Downarrow v}{(let\ [a\ (expr)]\ (expr2)) \Downarrow expr2[a \mapsto v]}$$

$$\text{if true}\ \frac{a \Downarrow true}{(if\ (a)\ b\ c) \Downarrow b)}$$

$$\text{if false}\ \frac{b \Downarrow false}{(if\ (a)\ b\ c) \Downarrow c)}$$

$$\text{cond}\ \frac{a \Downarrow false \quad c \Downarrow true}{(cond\ (a)\ b\ (c)\ d) \Downarrow c)}$$

$$\text{defn}\ \frac{expr\ [x] \Downarrow v}{(defn\ name\ [x]\ (expr)) \Downarrow name \mapsto v}$$

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

Chapter 5

Implementation and Testing

5.1 Section Heading

Chapter 6

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.

6.1 Section Heading

Chapter 7

Results/Evaluation

7.1 Section Heading

Chapter 8

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 1

(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

(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(
         (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
         (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)){
```

```
(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.