Beyond .*Script

Implementing A Language For The Web

Veit Heller

August 3, 2016

A thesis submitted for the degree of
B.Sc. of Applied Computer Science of
The University of Applied Sciences Berlin

For Meredith, Tobias and all the people who cope with me. Your undying support will not be forgotten.

Except where otherwise indicated, this thesis is my own original work.

Veit Heller
August 3, 2016

Abstract

This thesis presents and evaluates a port of the zepto programming language to the web. It aims to work as seamlessly with existing technologies as possible and is largely influenced by Revised^5 Report on the Algorithmic Language Scheme (R5RS), a standard of the Lisp derivative Scheme[1].

It is the result of over a year of independent research on zepto, a new programming language targetting various environments. The prototype described here and implemented for this thesis runs on JavaScript, a language found natively in many web browsers.

The central problem addressed by this thesis is the incoporation of a language runtime into the web ecosystem without the need for extensive code rewrites while supporting most features of the reference implementation of zepto, including large parts of the standard library. This also includes interoperability between the two languages.

It is also discussed inhowfar tooling for both JavaScript and zepto, most specifically their respective package managers, interoperate for building larger scale web applications.

[1] A deeper introduction into R5RS can be found in 2.2.

Contents

Abbreviations

API Application Programming Interface. 1, 22–25

AST Abstract Syntax Tree. 14

BSD Berkeley Software Distribution. 11

DOM Document Object Model. 1, 22, 23, 26

DSL Domain-Specific Language. 15, 30

EEP Erlang Enhancement Proposal. 10

FFI Foreign Function Interface. 6, 7, 22, 24–26

FRP Functional Reactive Programming. 8

GHC Glasgow Haskell Compiler. 6, 14, 21, 22

GUI Graphical User Interface. 8

IR Intermediate Representation. 14

JIT Just In Time Compiler. 6

LLVM Low Level Virtual Machine. 6, 13, 14

PEP Python Enhancement Proposal. 10

R5RS Revised^5 Report on the Algorithmic Language Scheme. ii, 9, 10, 16

REPL Read-Eval-Print Loop. 22

SRFI Scheme Request For Implementation. 10

W3C World Wide Web Consortium. 23

YUI Yahoo User Interface Library. 1

zeps Zepto Package System. 11, 18, 30, 36

ZPR Zepto Package Registry. 18

1. Introduction

> Controlling complexity is the essence of computer programming.
>
> *(B. Kernighan)*

1.1. Motivation

JavaScript has, since its inception, attracted a lot of controversy. This is rooted in various aspects of its design, from prototypal inheritance and the DOM to its operator precedence. Especially prototypal inheritance has been the root of a lot of discussions in the Computer Science community. It has the reputation of being counter-intuitive, though it is older than JavaScript, the first commonly known programming language that implemented prototypal objects being Self.

This and a few other design choices have prompted many programmers to develop wrapper libraries around almost anything that comprises the language. The most widely-used example of this is arguably jQuery, a library that abstracts over the DOM. But more arcane topics have been covered as well: there is, for example, the Yahoo User Interface Library (YUI), a now-abandoned project of Yahoo that aimed to abstract over web Application Programming Interfaces (APIs) but also comes with its own inheritance model. This model likens JavaScript prototypes to classical classes by introducing special functions and properties to the objects such as `extend` and `superclass`.

All of those libraries have shaped the way JavaScript has developed and in the upcoming revisions of ECMAScript[1], ECMAScript 6[2] and ECMAScript 7[3], a lot of conve-

[1]ECMAscript is the officially trademarked name of what application developers and browser vendors most commonly refer to as JavaScript.

[2]ECMAScript 6 was renamed to ECMAScript 2015 along the way but for the sake of clarity this thesis uses the more commonly know name of the revision. For an up-to-date version of that revision please refer to (ECMA International, 2015).

[3]Now ECMAScript 2017. For an up-to-date version of the current draft please refer to (ECMA International, 2016).

niences from third-party libraries have been adopted by the ”vanilla”[4] JavaScript canon. Among these features has been the widely controversial introduction of a class notation for JavaScript that makes prototypes feel more like classes (without breaking existing semantics).

1.1.1. Preprocessors & Transpilers

This and other features have been available to JavaScript developers for much longer than the latest iterations of the JavaScript language without them needing to rely on a possibly massive library - provided they use a preprocessor. One of the earliest exemplars of preprocessing for JavaScript, CoffeeScript, included lambdas, a shorthand for anonymous functions; pattern matching, a technique for destructuring data; and classes, all of which found their way into the latest ECMAScript revisions.

Preprocessing can serve a lot of different purposes apart from providing syntactic commodities to the programmer. Babel, for instance, is a relatively novel library for transpiling JavaScript that adheres to the new standards of ECMAScript back to older revisions, for the sake of backwards-compatibility. It also aims to provide small optimizations before the code reaches the compiler, such as constant hoisting, a compiler optimization technique that eliminates pure functions that always produce the same result in favour of global constants to reduce the overhead of function calls.

Transpilation from a different language is another increasingly popular way to use JavaScript as a platform. One could argue that CoffeScript is nothing short of a transpiler; but one could also argue against this idea, considering its only purpose is to compile to JavaScript. Other general-purpose languages such as Clojure and C++, originally developed to run on other platforms, have the option to compile to JavaScript through ClojureScript (Hickey, 2016) and Emscripten (Zakai, 2013) respectively.

Caveats

This thesis was inspired by the work done in the field of transpilation to JavaScript. Its goal is not to present yet another attempt at transpilation, but rather to rethink the way languages are incorporated into the web of today.

At their heart, all of the preprocessors and transpilers that target JavaScript are JavaScript, even if the syntax and semantics differ radically. There needs to be a clean

[4] A common name for JavaScript that refrains from leveraging paradigm-altering libraries.

mapping of the source language to JavaScript. Because of its flexibility, this is often a doable task, albeit not always desirable.

The value of having the same abstraction present at runtime as at compile-time is obvious if the language that is consider places strong emphasis on code mutability, such as in Lisp or Elixir[5].

1.2. Goals of this Thesis

The primary goal of this thesis is to present a novel approach at implementing languages for the Web. This is exemplified by a sample implementation of a non-trivial, pure and largely feature-complete functional programming language that has been tested in production systems.

Many reasons would speak for a procedural or objective-oriented language as a prototype. On one hand, the language itself could be more easily implemented in JavaScript if it is reasonably close to it. Another point that speaks against using a functional - and especially pure - programming language is the possibility of making interoperability harder because JavaScript is intrinsically impure and stateful.

But all of those reasons could also be read as argument *for* the implementation of a functional language. The difference showcases JavaScript's ability to express concepts foreign to the language in it. It also serves as a better test bench for more advanced implementation patterns.

This is especially true for those features of the base language that are not present in JavaScript and non-trivial to add to it from a user perspective. The features present in the target language implemented in this thesis but not in JavaScript most notably include Macros[6] and Continuations[7]. Those features were chosen especially because they are extremely foreign to JavaScript programming.

1.3. Structure of this Thesis

Chapter 2 examines related work in the field of cross-compilation into JavaScript and implementation of interpreters that are directly embeddable into larger systems. This includes desktop applications, game scripting engines and creative suites.

[5] As exemplified by Chris McCord in (McCord, 2015).
[6] The ability to rewrite code at parse or compile time, see 3.1.3.
[7] The control state of a program represented as a data structure within the code, see 3.1.4.

Chapter 3 gives an overview of the concept design and how the features are laid out to match the needs of both the goals of this thesis and the prototype itself.

Chapter 4 presents the system design and how the prototype integrates into existing web components.

Chapter 5 discusses the implementation, picking out different fundamental parts of the system and presents how they work.

Chapter 6 evaluates the prototype. This includes problems such as how well the integration of the system worked and how it compares to the reference implementation of zepto.

Chapter 7 gives a brief summary of what was done and gives an outlook to what might happen with zepto, both the desktop and the JavaScript version, in the future.

2. Related Work

This section aims to give a quick overview of work that has already been done in the field of transpilation to and language implementations on top of JavaScript. A few of the most important specimens have already been mentioned briefly in 1.1.1, although their relationship to this thesis have not yet been discussed.

2.1. Existing Projects

In this section, a brief overview over a short, not necessarily exhaustive list of transpilers from existing programming languages to and languages explicitly acting as a layer of abstraction over JavaScript shall be presented.

The aim of this section is not to present the reader with the syntax and semantics of every single language that is discussed, but rather equip them with a general overview of the ecosystem at the time of writing and how it correlates to the work presented in this thesis.

2.1.1. Transpilers

Transpilers are normally defined as compiling one high level language into another. They are often referred to as cross-compilers, although this term is imperfect as it also refers to compilation from one hardware platform or operating system to another.

Due to this ambiguity, the name *transpiler* was chosen to refer to this kind of system in the context of this thesis.

GHCJS

GHCJS - often hyphenated as GHC-JS - is a transpiler from the Haskell language to JavaScript. It is currently maintained by Luite Stegeman. It aims to "solve the JavaScript problem"[1]. It enables the user to compile any Haskell program to JavaScript

[1] A talk given by Stegeman bore this title, see (Stegeman, 2015).

instead of machine code by the means of inserting a custom compiler backend into the Glasgow Haskell Compiler (GHC) toolchain.

GHCJS is the compiler used in this thesis to transpile the existing zepto codebase to JavaScript. Its plug-and-play design, complete enough implementation of the Haskell programming langauge and relative maturity all played a role in the choice to use it as an implementation language[2].

Emscripten

Emscripten as discussed in (Zakai, 2013) aims to be a transpiler from Low Level Virtual Machine (LLVM) to JavaScript. Its primary focus is the translation of C and C++ source code to JavaScript.

Its use case is similar to that of GHC-JS, enabling the user to both "(1) Compile code directly into LLVM assembly, and then compile that into JavaScript using Emscripten, or (2) Compile a language's entire runtime into LLVM and then JavaScript, as in the previous approach, and then use the compiled runtime to run code written in that language" (ibid.), the latter of which sounding suspiciously like what zepto-js tries to achieve. There has been efforts to make the PyPy Just In Time Compiler (JIT)[3] work in the browser, and while it seems to have gained a fair bit of traction and work reasonably well[4], documentation on it is sparse and there is seemingly no academically viable source detailing its development and architecture.

ClojureScript

ClojureScript is a backend for the Clojure programming language that targets JavaScript. Initially developed by Rich Hickey, the author of Clojure, it is now maintained under David Nolen's lead (Hickey, 2016).

Being a Lisp, there are obvious syntactic similarites to zepto. Yet, as it is transpiled rather than interpreted directly in the browser, programming against it is quite difficult. Specifically, the way the FFI works is quite apparently** different, as ClojureScript has a lot of syntactic integrations that help embed JavaScript within it. **new sentence** all functions prefixed with a period are assumed to be JavaScript functions on the prototype

[2]Further discussion of the design and decision process can be found in 3 and 4.

[3]A JIT aimed to speed up Python programs by analyzing and compiling them, described in (Bolz et al., 2009).

[4]A live demonstration of how it works can be found on its website, accessible under https://pypyjs.org.

of the object provided as first argument. JavaScript values can be directly accessed through the js namespace. Zepto's design is quite different, as is discussed in 5.2.2.

```
; ClojureScript provides syntactic abstractions over
; the embedding of foreign code.
(.getElementById js/document "body")

; An equivalent zepto call, with strings
(js "document.getElementById(\"body\")")
; OR, if a variable is accessed
(js (++ "document.getElementById(\"" body "\")"))
```

Listing 1: A comparison of the FFI of JavaScript in zepto and ClojureScript.

2.1.2. Abstraction Languages

The languages mentioned here are languages that purely abstract over JavaScript, choosing it as their primary backend. This differs from the languages discussed in 2.1.1, which are general purpose languages with the option to compile to JavaScript if desired by the developer.

This list is not designed to be exhaustive, but rather aims to inform about recent developments in web programming.

CoffeeScript

As mentioned int 1.1.1, CoffeeScript was one of the first transpilers that targetted JavaScript when it first appeared[5]. Many of the constructs have been adopted by JavaScript by now, both in terms of syntax and semantics. CoffeeScript is similar to JavaScript in that it is a procedural, prototypal language. No semantic features have been added or removed from JavaScript, making it relatively dissimilar to the programming system presented in this thesis.

[5]The first git commit dates back to December 13th, 2009. The first public release happened shortly after that.

Elm

Elm as it is presented in (Czaplicki, 2012) is a language centered around Functional Reactive Programming (FRP)[6] with a special regard to browser-based Graphical User Interfaces (GUIs). Formulated by Paul Hudak and Conal Elliott in a 1997 paper, it borrows its syntax largely from Haskell and is a pure, functional programming language. Notable features include GUI-centric tooling, such as a ”time-traveling” debugger that caches the state of a program at any given time, allowing for rewinding, forwarding and jumping.

Elm's sole similarity to the language discussed in this thesis is that it is a functional language for the Web. However, it compiles to JavaScript, HTML and CSS, once again falling short of actually getting the code to the browser. It is also very centered around the aforementioned FRP, whereas the runtime presented in this thesis aims to be a general-purpose programming environment.

PureScript

PureScript is, much like Elm, a functional, pure programming language that borrows a lot of its syntax from Haskell. It aims to compile into human-readable JavaScript and make general purpose programming for the web in a strongly typed, functional style possible.

It is different from GHCJS insofar as it does not provide compatibility with Haskell code, although many libraries and functions are relatively similar. See 2.1.2 for an example.

As it shares a lot of features with Elm, it also largely shares the differences to the system presented in this thesis.

[6] An in-depth explanation of FRP is outside of the scope of this thesis, but it is a programming style (Elliott and Hudak, 1997).

```
-- | Extract the first element of a list (as implemented in the language standard library).
-- | The Haskell version is unsafe and will throw an error
-- | if it encounters an empty list.
head :: [a] -> a
head (x:_) =  x
head [] =  badHead -- an exception function

-- | the list data type in PureScript is safe by design and
-- | the function returns a Maybe monad.
head :: forall a. List a -> Maybe a
head Nil = Nothing
head (Cons x _) = Just x
```

Listing 2: A juxtaposition of a simple function in Haskell and PureScript.

2.2. Existing Standards

When developing a programming language derived from the Lisp family, one builds on top of almost 70 years of development, formalization and research. Of course many of the standards that were formed during that time are now obsolete. Nonetheless, the effort of standardizing the many languages and implementations is still thriving.

Over the years, two main categories of languages have developed: Common Lisp and Scheme. While the syntactic proximity remains, they differ widely in concepts. 2.2 shows the differences between those language families.

Zepto is a Scheme derivative loosely based on R5RS (Kelsey et al., 1998), the most widely implemented standard of Scheme. It takes from it a lot of stnadard library functions, syntax definitions and continuations, but aims to introduce custom namespaces to provide a cleaner, more modular way of defining software libraries[7].

The reason why a Scheme derivative was chosen over an implementation of Common Lisp can be found in the feature set: hygienic macros and continuations are valuable abstractions that guarantee that components work together more seamlessly. An example of code that works as expected in Scheme but not in Common Lisp can be found in **??**. It implements a simple if a bit contrived version of binary `or`, in Common Lisp and

[7]This module system will be explained in 3 in depth.

	Macro Keyword	Macro Dispatch	Hygienic Macros	Continuations	Standard
Scheme	define-syntax	pattern matching	present	present	ANSI Common Lisp
Common Lisp	defmacro	argument passing	not present	not present	RNRS

Figure 2.1.: A comparison between Common Lisp and Scheme

in Scheme. In Common Lisp, the variables defined in regular code will potentially be shadowed by the ones generated during macro expansion where in Scheme they reside in different contexts and thus are hygienic. This makes the system more intuitive and eliminates a class of potential bugs and is thus more desirable.

While R5RS is certainly a valuable standard and provided a lot of very useful information for the development for zepto, the language is not as closely tied to the standard as most other implementations. This allowed for the development of zepto-js to progress relatively unencumbered by incompatibilities, although one goal of the development effort was not to break any existing syntactic or semantic ties with R5RS, as this might potentially have broken a big part of the standard library, one of the major arguments for zepto as a language.

2.2.1. Scheme Requests For Implementation (SRFIs)

An important argument for adhering to the R5RS standard are Scheme Requests For Implementation (SRFIs). They are proposals of the Scheme programmer community to enhance the standard libraries of the implementations through the additions of features, much like Python Enhancement Proposals (PEPs) for the Python programming languages and Erlang Enhancement Proposals (EEPs) for Erlang. However, the differences that sets the former apart from the latter two is the inclusion of a reference implementation[8]. Zepto currently supports 5 of these requests by default, one of which implements a core feature of the language, descriptive custom data types.

It proved to be a major convenience in the past that the implementation of these requests could be ported with relative ease. By moving away even more from the Scheme standards, this process would undoubitably be affected in a negative manner. Thus, for both the reference implementation and its JavaScript equivalent, it was decided that a

[8]This is not required, but part of the ”good tone” of the community and SRFIs lacking them are relatively rare.

core set of primitives must be available under all circumstances to allow for the cross-fertilization of Scheme and zepto.

```
; the Common Lisp version
(defmacro my-or (x y)
 `(let ((tmp x))
   (if tmp
     x
     y)))

(my-or #f #t)
; works (yields true), expands to:
(let ((tmp #f)) (if #f #f #t))

(let ((tmp #t))
 (my-or #f #t))
; does not work (yields false), expands to:
(let ((tmp #t)) ((tmp #f)) (if #f #f tmp))

; the Scheme version
; it will work as expected in both versions
(define-syntax
 (syntax-rules
   ((my-or x y)
     (let ((tmp x))
       (if tmp
         x
         y)))))
```

Listing 3: Common Lisp Macros vs. Scheme Macros

2.2.2. Unix

While not necessarily a standard, Unix and its guidelines do affect zepto in an indirect way as much of zeptos tooling, such as the zeps are using Unix utilities and system calls internally. This is expected to change in the future, but so far all the developers that have worked on or with zepto[9] have used Unix or Unix-like systems such as MacOS X or Berkeley Software Distribution (BSD). This is less of a problem in the browser context because many of the packaging and versioning tools are run from the command line

[9]At least to the knowledge of the author.

before the application is distributed, but it is important to keep in mind that some of the tools rely on a specific Operating System when developing zepto programs.

3. Concept Design

Practicality beats purity.

(T. Peters—The Zen of Python)

The concept of a version of zepto that would run on JavaScript arose rather naturally from the work done on zepto as a backend-agnostic, run-everywhere concept language. Concurrently to the work on a nanopass compiler that targets LLVM and Erlang but supports pluggable backends by design, efforts have been made to bring the language into the browser setting. After preliminary work on a compiler backend that emits JavaScript that was discarded in the early stages of its development for the reasons stated in 1.1, the idea presented in this thesis emerged.

In the following, a brief overview of the design of the concepts used in the port and how they impacted the construction of the system shall be given. A system-based design overview shall be given in 4.

3.1. Construction Design

The value of integrating zepto into existing web infrastructure lies in the addition of a host of new features that are not present natively in JavaScript and hard to implement in a stable manner[1].**Make sentence into two or eliminate clauses; restructure to eliminate intro** The goal of the prototype was thus to expose all of the features that define zepto in the JavaScript implementation as well.

A brief overview over why a Lisp was chosen for this thesis and, more specifically, why zepto, shall be given in the following.

[1]While stability is not guaranteed by the current version of zepto, it is tested and deployed in production systems and shows to function rather reliably.

3.1.1. Lisp

A common saying among programming language designers is that every programmer has written their own implementation of Lisp.[2] There are a lot of different implementations of Lisp in the wild, even ones that compile to JavaScript, such as ClojureScript, the backend for Clojure referenced in 2.1.1.

The simplicity of the language on a parsing level is often cited as the main reason for its popularity as a language to implement. A simple Lisp can be implemented in less than one hundred lines of code if no intermediate representation is generated. This is made possible by the unique property of Lisp of enclosing every statement in parentheses, where the first element within those parentheses is the statement name and the other elements are the arguments. It can be evaluated straight from a textual level, because things such as operator precedence and statement amiguity do not exist. In regular Lisp as specified in the initial paper by John McCarthy (McCarthy, 1960) only a handful special forms exist to allow not only for Turing-completeness, but also for expressiveness and elegance.

3.1.2. zepto

As explained in 2.2, Zepto is a new Scheme implementation that aims to be as small as possible, to be able to target a lot of different backends and simplify the process of writing new backend code. Currently, LLVM and Erlang Core[3] bindings are under development, the reference implementation is a simple interpreter that interprets code directly from the Abstract Syntax Tree (AST). This is slow but ensures a small interpreter size[4]. The compilers are written directly in zepto itself.

The small code base makes zepto a good target for porting it to the web. Further, because it is written in Haskell the code base was expected to be possibly almost entirely compilable to JavaScript using GHCJS, the transpiler mentioned in 2.1.1, a backend for the GHC targetting JavaScript instead of native code. It offers many advanced compilation features such as inlining of JavaScript into the Haskell code base using a

[2]The README of the femtolisp project even goes so far to claim ”Some programmers’ dogs and cats probably have their own lisp implementations as well.” (Bezanson, 2016).

[3]Erlang Core is the Intermediate Representation (IR) of Erlang code before it is complied. Resources and documentation about it are sparse, it mostly seems to exist inside the BEAM’s implementation. A small paper (Carlsson, 2001) that describes its’ basics was used to implement the compiler from zepto.

[4]The entire codebase is only about 4000 lines of Haskell code.

technique called quasi-quoting, where a special character sequence delimits the inlined code, much like regular quotes. This tool set was expected to make the work of porting an existing language to the web as simple as possible.

Of course there are other reasons to use a functional language as an example for how to bring a language into the browser. With both syntax and semantics differing strongly from JavaScript, this example enables languages more closely related to JavaScript to eventually make their way into the browser.

3.1.3. Macros

Macros are a mechanism for rewriting code at compilation time. As discussed in 2.2, zepto's macros are hygienic, which prevents name collisions with code at runtime and eliminates a class of particularly nasty bugs. Macros are evaluated in a step between the parser and the actual runnig or compilation of code called macro expansion, where they rewrite parts of the source code that reference it according to its specifications.

Because Scheme macros leverage pattern matching and argument overloading by default, they allow for syntactic extensions of the language. An example for a simple yet interesting case is made in 3.1.3, where a simple syntactic rule set for list comprehensions is defined, not unlike a Domain-Specific Language (DSL), albeit a very small one. This particular list comprehension is syntactically close to the same construct in Haskell, although the semantics should be relatively clear at a close inspection and are, for the sake of example, a bit simple than in the language it is inspired by.

```
; this defines a macro of the name listcomp
(define-syntax listcomp ; the name
  (syntax-rules (<- |) ; the set of rules and special tokens to respect
    ((_ expr | elem <- elems) ; the pattern to match where
                              ; the underscore character is a wildcard
     (map (lambda (elem) expr) elems)))) ; the pattern is simply rewritten to a call to the map function

; add1 to every x in the list
(listcomp (add1 x) | x <- [1 2 3 4]) ; => [2 3 4 5]
(listcomp (add1 x) | x <- (range 4)) ; => [1 2 3 4]
```

Listing 4: Defining & using a macro in zepto

It should be noted that zepto supports list comprehensions by default and they are both a bit more concise and more powerful - as they allow for the filtering of elements from the source list - than the version defined here.

3.1.4. Continuations

Continuations are a datatype representing the state of a program at the time of request, including environments, definitions and execution path. By manipulating it and injecting it into different contexts, it becomes a powerful tool to switch back and forth between multiple threads of execution, without actually creating or managing threads. A practical application of coroutines was for instance presented in (Haynes et al., 1984), where it was used to implement a fully formed coroutine system that was capable of starting, stopping and pausing execution of multiple coroutines at once without the need for actual threading.

An exhaustive explanation of continuations cannot possibly be given in the scope of this thesis, for further information about this concept refer to (Reynolds, 1993).

3.2. Additional Features

There are features in zepto that are not present in Scheme as specified by R5RS, such as the module system and the extensive standard library. In this section, a brief list of the features and libraries that discriminate zepto from "regular" Scheme shall be given.

3.2.1. The module system

Zepto implements a custom module system that is completely defined in terms of the language itself. A small code example of different actions in zepto's module space are given in 3.2.1. This allows for namespaced, conditional and renaming imports, a feature missing from standard Scheme[5]. This approach has a few upsides that all boil down to the module system being available in the user namespace; monkey-pacthing and mocking objects is possible very easily, easing for instance testing and fixing flaws in the program at runtime. It is also relatively small and simple, at about 130 lines of macro-backed code, which makes it fairly maintainable.

However, this approach comes not without its drawbacks. Most notably, relying on a module system that works at runtime requires more complex compilation and a dispatch

[5] Although the upcoming R7RS standard will most likely include such a mechanism.

system to be present while the program is running. It also requires global data in the form of a special variable *modules* holding the mapping of names to values, a concept that is unusual for a functional programming language and might hinder the compilation into other representations. Of course global data is also normally either not thread-safe or relatively slow to use if it requires some sort of synchronization mechanism - which is currently not a feature that zepto implements.

It has been decided that this is an acceptable tradeoff for zepto as it currently stands, but the system might be changed in the future.

```
; this defines a module of the name "mathematics"
(module "mathematics"
  (export
    `("add" ,add)
    `("substract" ,substract)
    `("multiply" ,multiply))

  ; the actual worker function (not exposed)
  ; applies a function to a list of arguments
  (doop (op args)
    (apply op args))

  ; the exported functions; essentially just wrappers
  ; around doop
  (add (lambda arguments (doop + arguments)))
  (substract (lambda arguments (doop - arguments)))
  (multiply (lambda arguments (doop * arguments))))

; will return a reference to the function
(import "mathematics:add")
; so that it can be bound to a name
(define add (import "mathematics:add"))
; or called directly
((import "mathematics:add") 1 2)

; import all under the namespace "mathematics"
(import-all "mathematics")
; import all under the namespace "mt"
(import-all "mathematics" "mt")
```

Listing 5: Defining & using a zepto module

For the case of JavaScript, this part of zepto's implementation proved not to be a problem and worked without further tweaking.

3.2.2. The standard library

Zepto comes with a fairly extensive standard library that includes a wide variety of utilities spanning such diverse topics as cryptography[6], testing, monads, parsing command line arguments and a parser combinator. An exhaustive list of the modules found in the zepto standard library at the time of writing can be found in A.

This is fairly unusual for Scheme implementations insofar as many of them come with a fairly minimalistic set of libraries that is often largely based on SRFIs exclusively. Two notable exceptions are Chibi Scheme[7], and Racket, the distributions of which are bundled with an even bigger standard library. This might be due to the fact that Alex Shinn, the main developer behind Chibi Scheme, is very active in the Scheme community and authored a few SRFIs of his own and Racket is a language supported by the Computer Science faculties of multiple universities and is the result of over 20 years of research.

The stability of the libraries that come bundled with zepto varies and is not ensured until the reference implementation reaches version 1.0.0. However, many of these libraries have at least been used by other libraries in the zepto ecosystem and are thus the subject of continuous scrutiny.

The standard library as a concept in zepto is thought to be a rather minimal, but usable set of primitives to get the developer started. Once they feel comfortable in the programming environment, it is suggested they start working with zeps.

3.2.3. zeps

zeps is the package manager for zepto. It is capable of installing and managing packages from Github and the Zepto Package Registry (ZPR) through the command line. It is undoubitably the biggest coherent piece of code written entirely in zepto and it is capable of intalling and updating itself. At the time of writing, at least 40 packages have been published with zeps, including a framework for writing web servers and a Redis database

[6] As found in the rsa module, which implements RSA key generation, signing, validating and en- and decryption.

[7] The source code for Chibi Scheme and its standard library can be found on Github under https://github.com/ashinn/chibi-scheme.

client. It is only able to run on Unix-like systems or systems exposing at least a Unix-like shell.

The authentication system of zeps on the ZPR is RSA-backed, which also allows for the signing of packages for the user's convenience. This feature is not available when installing from Github.

The package system aims to be a cohesive, intuitive entity for installing, removing, creating, managing, testing and updating packages[8]. It includes facilities for creating sandboxed environments, making dependency management of multiple systems with possibly conflicting dependency trees possible. It can also be used to distribute zepto-based command line tools and its ability to bootstrap packages is scriptable, allowing for plugins to decide what files should be generated on the creation of a new package.

It has proved to be useful in workshops and user feedback sessions to make sure all of the attendees can easily manage their first projects.

[8] A list of all the available commands is given in B.

4. System Design

Practicality beats purity.

(T. Peters—The Zen of Python)

4.1. Integration into the Web Ecosystem

5. Implementation

> It always takes longer than you expect, even when you take into account Hofstadter's Law.
>
> *(Hofstadter's Law)*

The implementation philosophy of the port presented in this thesis has always been to reuse as much code from the reference implementation as possible. This guided the flow of design choices down a rather natural path and kept the implementation described here fairly short and trivial.

5.1. Description of the Toolchain

The tooling uses GHCJS, which is a backend for the GHC compiler that targets JavaScript rather than native code (TODO: remove this from introduction). This makes cross-compiling the code base to JavaScript a rather simple undertaking.

GHCJS offers many advanced features such as inlining of JavaScript into the code base using a technique called quasi-quoting, in which a special character sequence delimits the inlined code, much like regular quotes. This tool set was expected to simplify rewriting the code bits that needed adjustment.**verb tense: does it no longer simplify rewriting code?**

The management of pure JavaScript sources is another important feature. GHCJS provides management of these sources akin to the management of bits of C code that should interface with Haskell in a regular GHC project.**wordy** The build file includes a segment called js-sources, which ensures that the JavaScript in those sources will be included in the final compilation. The informal convention seems to be to put the files that should be included into a directory called jsbits (a hat-tip to the GHC convention of putting C sources into cbits).

The code emitted by GHCJS is conservative in feature use. This is an important feature if older clients with possibly even obsolete browsers want to access web pages supported by zepto code, rare as they might be.**wordy**

All these conveniences do not spare the programmer from the actual programming process, of course. While GHCJS seems like a well-maintained project one has to keep in mind that it is not backed by a consortium as GHC is, and the current maintainer seems to do this work completely in his spare time. This leaves us with the notion that the project could be abandoned at any point and being too dependent on its' more unusual features could lead to a product that is unnecessarily hard to maintain. **Suggested rewrite: This gives users the impression that the project could be abandoned at any given moment. Therefore, being too dependent upon its more unusual features could lead to a product that is unnecessarily difficult to maintain.**

All of this considered, a preliminary analysis revealed that the two main conditions for the success of this thesis were given: the code emitted by GHCJS is fast, solid and reasonably compact and the newest features of both JavaScript and Haskell were supported[1]. **Suggested rewrite: Taking all of this into consideration, a preliminary analysis revealed two main conditions for the success of this thesis: 1) the code emitted by FHCJS is fast, solid, and reasonably compact, and 2) the newest features of both JavaScript and Haskell were supported.**

5.2. Description of the Implementation

As predicted in 1.1, the code base of zepto could be reused almost in its entirety. What had to be rewritten was mostly related to the startup of the interpreter, because the regular paths into the code - either via a script being passed into it or launching an interactive Read-Eval-Print Loop (REPL)[2] - were unavailable in the browser context. Instead, a way of passing the sources from within `script` tags needed to be found. Further customizations include a FFI to enable better cross-evaluation of JavaScript and the adaptation of existing APIs, such as the DOM.

[1] The newest version of GHC at the time of writing, version 8.0, gained support while this thesis was in the works.

[2] A REPL is an interactive code evaluation environment. Code is typed into a prompt and immediately evaluated. The convenience of such a short feedback loop is often used in the context of scripting languages and shells.

5.2.1. The script tag

Initially, a DOM node walker was considered, but rejected relatively early because of two reasons: firstly, it introduced a layer of complexity from within JavaScript code that would have likely made it brittle and hardly portable. Secondly, it would require a walk of the nodes every time a DOM element is inserted or replaced, which is a common occurence in modern interactive web applications.

In November 2015, the World Wide Web Consortium (W3C) specified **word choice?** an API that simplified this process for the programmer. Within their specification of the DOM4 (World Wide Web Consortium, 2015), the fourth specification of APIs for the Web, an object called `MutationObserver` is included which is able to register for DOM manipulations. Its main function will be triggered whenever a change occurs within the DOM part, to which it is registered to listen. **rephrase as you wish, but to should not be at the end of the sentence**

This simplifies the implementation of a listener to DOM events a great deal. Only minimal programming is required to configure the listener and to filter out all the nodes that are not `script` nodes of the type `text/zepto`[3].

A problem left ignored with this method was that nodes inserted before the listener started.**awkward, rephrase** This was resolved by singling out all the `script` tags present before the listener starts and applying the same filter/evaluation function to all of them.**verb tenses** This also ensures that they are executed before any additional code (and possibly dependent) is passed into the zepto object.

The code was then included in the `zepto` singleton, which is the global interpreter object used for the management and interactivity of the zepto interpreter.

[3] This was chosen in analogy to the existing `text/javascript` node type.

```
// the initial observer and the function it takes
zepto.observer = new MutationObserver(zepto.handleMutation);

// this function will get a list of mutations and apply handleDom to them
zepto.handleMutation = function(mutations) {
  mutations.forEach(mutation => {
    mutation.addedNodes.map(zepto.handleDom);
  });
}

// evaluate if it is a text/zepto node
zepto.handleDom = function(node) {
  if (node.nodeName != "SCRIPT" || node.type != "text/zepto") {
    return null;
  }
  return zepto.eval(node.innerHTML);
}

// execute this on startup
window.onload = () => {
  let scripts = document.getElementsByTagName("script");
  scripts.map(zepto.handleDom);
}
// the extra arguments signify recursive listening
zepto.observer.observe(document, {childList: true, subtree: true});
```

Listing 6: The final mutation observer code (simplified)

5.2.2. The FFI

The FFI is a central part of the port. If it were unusable, none of the browser's capabilities could be used from within zepto, thus rendering the effort of bringing zepto into the browser effectively useless. The APIs of the Web are a big part of what it means to program for the browser, after all.

An initial sketch of the programming interface was extremely simplistic: a call to the function `js` could be called with a string as argument, representing the textual representation of the JavaScript program that should be run. It was piped to the JavaScript function `eval` and the function returned an affirmative truth value. Quasi-quoting larger blocks of JavaScript was also possible.

Of course this is unusable. **word choice, see above** The missing return value makes any effort of talking to an API impossible, as one could never yield any results.**rephrase clause 2** A different kind of return value is needed.

The obvious but most challenging to implement solution would be to infer a fitting zepto type for every return value in JavaScript and return a result depending on that.**Suggested rewrite: The most obvious solution – albeit the most challenging – would be to infer a fitting zepto type for every return value in JavaScript and return a result depending on these paramaters.** While this could be seen as a rather elegant solution, it comes with its own set of caveats and exceptions, as the mapping between JavaScript and zepto values is not always obvious. A JavaScript object has too many properties that get lost in the process of translating it to zepto as to make it intuitive.

```
; this would return an integer
(js "1 + 1")

; this would return a hashmap
(js "{key: \"val\""})

; this is problematic, because it will return an object
(js "new Error()")
```

Listing 7: The ideal FFI

Another problematic point is the implementation of JavaScript values in GHCJS.**Suggested rewrite: The implementation of JavaScript values in GHCJS is another point of complication.** They are opaque datatypes, aliases for addresses and byte vectors. While zepto supports byte vectors and pointers, they are hardly a good representation for semantically rich prototypes as they only offer a glance into the underlying implementation of the JavaScript engine. While it is true that GHCJS itself provides methods for type coercion, these methods are crude and possibly error-prone.

A simpler method but still mostly sensible arose: returning the string values of all of the values returned.** While this places the burden of coercion on the programmer, it also gives them the power to make their own decisions of how to deserialize values. Functions for deserializing the most common datatypes are included in the standard library of the JavaScript implementation of zepto, to aid the programmer in the process of finding the right methods of getting a value out of the FFI.

This still does not solve the problem of helping manage classes. It does, though, empower the programmer to find their own ways of serializing on the JavaScript side and deserializing on the zepto side to preserve the information they need in their specific programming context.

All of this needs an additional layer of abstraction to avoid unnecessary boilerplate, but it is stable enough for most purposes that zepto in JavaScript was used for yet.

```
; the function string->number is a standard zepto function
(string->number (js "Math.pow(2, 32)"))

; this is an example of how to resolve the earlier problem:
; override the prototype of the object to return the value that is needed
(js "Error.prototype.toString = function() { return this.message; }")
(error (js "new Error(\"fatal error occured\")"))
```

Listing 8: The final form of the FFI

Implementing the JavaScript to zepto FFI was much simpler, as the interpreter is defined within the JavaScript environment. A call to the `eval` function of the `zepto` object with a string as argument will return in the execution of this piece of code and the return the textual representation of the zepto object so that the entire communication between the languages is string-based.**rephrase; missing verb? make two sentences w/ break after "piece of code".**

5.2.3. The DOM

After building the FFI, it was possible to implement the entire communication with the DOM in terms of calls to foreign functions and the parsing of their return values. This allows for a stable library, because it is unintrusive and does not interfer with existing JavaScript constructs.

Existing zepto-js projects often find it convenient to write a hybrid mix of JavaScript and zepto code that calls each other at certain points. This is, however, probably not advisable at the layer of libraries or utilities, because it is at risk of getting in the way of the job.

```
(module "dom"
 (export
  `("create" ,create)
  `("insert" ,insert)
  `("get" ,get))

 (loads "html")

 (update-dom! (lambda (node contents)
  (js (++ "document.getElementById('" node "').innerHTML = '"
        (create contents)
        "';"))))

 (create (lambda (tree)
  ((import "html:build") tree)))

 (insert (lambda (context tree)
    (update-dom! context tree)))

 (get (lambda (node)
  ((import "html:parse")
    (js (++ "document.getElementById('" node "');"))))))
```

Listing 9: A minimal DOM module

5.2.4. Language definitions

One of the most advanced features is language definitions, inspired by the parser macro system in Racket (Flatt, 2011). While this is handled fairly differently in zepto than in Racket, the syntax of the resulting language is compatible. Zepto handles the custom parser step with a dispatch at the time of file loading, which enabled the system to be implemented without changes to the core language, completely in zepto itself. While this was designed with portability in mind and thought to ease the transition to different backends, it actually proved to complicate matters in the case of the JavaScript port. Here, the module loading system is not present due to the absence of files in the traditional sense.

```
; json-lang.zp
(load "json/json")

(zepto:implements-lang json:parse "json")

; an example for a file that can now be normally loaded
#lang json
{
 "hello": "json"
}
```

Listing 10: An example language definition that allows for inlining of JSON code.

As the detection of what to parse is handled from within JavaScript - by the MutationObserver mentioned in 5.2.1 - the first implementation of the dispatch mechanism was written entirely in JavaScript. This proved to make the dispatch mechanism relatively clumsy, because registering additional languages had to be done in JavaScript, making this feature a JavaScript rather than a zepto feature. While building a parser dispatch mechanism for JavaScript would certainly also make for an interesting experiment, it is out of the scope of this thesis and would not really qualify as something unique about zepto[4].

[4] The code was salvaged for posteriority and can be found on Github under the name of js-parse-dispatch.

6. Evaluation of the Prototype

When I'm working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.

(R. Buckminster Fuller)

6.1. Seamlessness of Integration

6.2. Test Against Standard Implementation of Zepto

* added: ffi

  * removed load statement, REPL functionality

  * inserting libraries?

7. Summary and Outlook

> Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.
>
> *(Isaac Asimov)*

The prototype presented in this thesis was never expected to be a replacement for technologies that are currently making the web what it is. It is a simple experiment that happens to work well enough to power less than a handful actual production systems, a blessing for the development of zepto. The use cases provided real insight into the descripancy of how the program works and how it should work. It is also a liability going forward, as users do not generally appreciate breaking changes.

The software system certainly proved that it is possible to make other languages work on the web, an idea that is nasceant but, by the hopes of those involved in the production of zepto-js, one that might become a trend going forward.

Other developments in the zepto ecosystem are just as exciting. There have been efforts to port zepto to a few different platforms, all of which are far from finished, but interesting to follow along nonetheless. The compiler and parser have both gained languages, some of them usable - the aforementioned compiler backends for LLVM and Erlang come to mind, and the web server framework that uses a DSL to simplify writing request handlers - , some of them born purely out of the desire to toy around with the language - parsers and cross-compilers for a handful of esoteric programming languages have been written, including Brainfuck, Io and Iota.

The development of zeps has surely contributed to the growth of zepto and it has fueled the development of zepto-js. It is tested on real systems and has withstood the scrutiny of colleagues and contributors. By the definition of the development team, however, it is not yet ready for a big public release, hence the timid version number of

0.0.7. The road to go might be not as long as this number suggests, but it certainly is long enough to merit a healthy dose of defensiveness.

Another big development in the zepto community has certainly been a series of classes and workshops. They, too, helped route the development of zepto down a path of usability, self-awareness and constant reevaluation.

TODO

8. Conclusion

There have been voices that insisted that the web need be fixed since its inception. Most disagreements over its inner workings have been philosophical and are a matter of ongoing dispute. While the web certainly can be improved - like any system, at any time - I do not believe it is fundamentally flawed. The technologies that power the current World Wide Web, from switches and cable technologies to protocols and development frameworks, have shown to be robust enough for a large part of the world to interconnect. We steadily adapt the way we work with it as trends and paradigms emerge and evolve.

One of the major redeeming qualities of the fundamental philosophies of the World Wide Web has certainly been its ability to adapt to changes. Standardization of web technologies has a reputation of being slow, but in terms of design processes on a global scale, it is actually a reasonably fast process.

Zepto is a young language that has adapted this mindset - changes rapidly and a stable release is yet to be announced. With this thesis, an important step towards reaching a fulfillment of its design goals has been made.

Many questions that this thesis has addressed remain unanswered. Even with that in mind, I hope that my work on it has asked questions that are worth being asked and that the technologies presented in this thesis will allow for more technologies that come to the web, so that an even more heterogeneous, expressive and inclusive system can emerge, one that caters to different aesthetics, philosophies and mindsets.

I would happily welcome any contributions towards that end, both related to and unencumbered by the technology that is zepto.

A. List of Modules in the Zepto Standard Library

Libraries that are loaded by default are denoted by *(dflt.)*.

argparse A command line argument parser

ascii An ASCII art modul

bench A simple benchmarking and timing library

calculus A library that implements combinators of the lambda calculus

char*(dflt.)* A library of functions for working with characters

cl A library that exports standard Common Lisp functions

data A library that exports lazy data types, such as queues, deques and streams

datetime A library for working with and formatting dates and times

delay*(dflt.)* A utility library for delaying computations

infix A library that translates infix mathematical expressions to the appropriate prefix form

io*(dflt.)* A library of functions for Input and Output

json A JSON parser and renderer

keywords*(dflt.)* A library that allows for keyword arguments as found in Python

marsaglia A library of functions for cryptographically strong Pseudo-Random Number Generators

math*(dflt.)* A library of functions for mathematical purposes

minitest A testing library

module*(dflt.)* Zepto's module system

monads A library of monads and monadic computations

parsecomb A parser combinator library

pointfree A library for defining functions in point-free style

querystring A querystring parser and renderer

random*(dflt.)* A library of functions for cryptographically insecure random number and data generation

rsa A RSA library

slugify A library for generations slugs

sort*(dflt.)* A sorting library

srfi*(dflt.)* A library of SRFIs

statistics A statistics library

struct*(dflt.)* A library for generating structs with the appropriate functions declaratively

zpbash A BASH library

zpcllections*(dflt.)* A library of functions for defining and working with collections

zpcont*(dflt.)* A library of functions for working with continuations

zpconversion*(dflt.)* A library of functions for converting between datatypes

zperror*(dflt.)* A library of functions for working with errors

zpfile*(dflt.)* A library of functions for working with files

zpgenerics*(dflt.)* A library of functions for defining and working with generic functions

zphash*(dflt.)* A library of functions for working with hash maps

zplist*(dflt.)* A library of functions for working with lists

zpnumbers*(dflt.)* A library of functions for working with numerical values

zpstring ***(dflt.)*** A library of functions for working with strings

zpvector ***(dflt.)*** A library of functions for working with vector

zpversion ***(dflt.)*** A library of functions for working with the current zepto version

B. List of zeps commands

This list can also be found in the README of zeps under https://github.com/zeps-system/zeps.

test Runs the module tests.

t shortcut for test.

search Search for packages matching a search term (on the ZPR).

sandbox Create/destroy a sandboxed zeps environment

run Run the module entry-point, without installing it

repl Launches an interactive shell with a certain module preloaded.

remove Removes a package.

rm shortcut for remove.

register Registers a package.

r shortcut for register.

new Bootstraps a new package.

n Shortcut for new.

keygen Generates a new RSA key for zeps.

install Installs a package.

i Shortcut for install.

help Interactive help on getting started

readme Prints zeps' README

References

Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman (2006). *Compilers: Principles, Techniques, and Tools*. Pearson Eudcation, Inc.

Bezanson, J. (2016). *The FemtoLisp programming language*. URL: https://github.com/JeffBezanson/femtolisp (visited on 01/08/2016) (cit. on p. 14).

Bolz, C. F., A. Cuni, M. Fijalkowski, and A. Rigo (2009). "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. Genova, Italy: ACM, pp. 18–25. ISBN: 978-1-60558-541-3 (cit. on p. 6).

Carlsson, R. (2001). "An introduction to Core Erlang". In: *In Proceedings of the PLI'01 Erlang Workshop* (cit. on p. 14).

Czaplicki, E. (2012). *Elm: Concurrent FRP for Functional GUIs*. Senior Thesis (cit. on p. 8).

ECMA International (2015). *ECMAScript®2015 Language Specification*. URL: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf (visited on 12/07/2016) (cit. on p. 1).

ECMA International (2016). *ECMAScript®2017 Language Specification*. URL: https://tc39.github.io/ecma262/ (visited on 12/07/2016) (cit. on p. 1).

Elliott, C. and P. Hudak (1997). "Functional Reactive Programming". In: *Proceedings of the second ACM SIGPLAN international conference on Functional programming* (cit. on p. 8).

Flatt, M. (2011). "Creating Languages In Racket". In: *acmqueue* 9 (11) (cit. on p. 27).

Fogus, M. (2013). *Functional JavaScript*. O'Reilly Media.

Haynes, C. T., D. P. Friedman, and M. Wand (1984). "Continuations and coroutines". In: *In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (cit. on p. 16).

Hickey, R. (2016). *The ClojureScript Wiki*. URL: https://github.com/clojure/clojurescript/wiki (visited on 14/07/2016) (cit. on pp. 2, 6).

Kelsey, R., W. Clinger, and J. R. Editors (1998). "Fifth Revised Report on the Algorithmic Language Scheme". In: *ACM SIGPLAN Notices* 33.9. Ed. by R. Kelsey, W. Clinger, and J. Rees, pp. 26–76 (cit. on p. 9).

McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Communications Of The ACM* (cit. on p. 14).

McCord, C. (2015). *Metaprogramming Elixir: Write Less Code, Have More Fun.* The Pragmatic Programmers, LLC (cit. on p. 3).

Parr, T. (2010). *Language Implementation Patterns.* The Pragmatic Programmers, LLC.

Queinnec, C. (2003). *Lisp in Small Pieces.* Cambridge University Press.

Reynolds, J. C. (1993). "The Discoveries of Continuations". In: *Lisp and Symbolic Computation* 6.3-4, pp. 233–248 (cit. on p. 16).

Stegeman, L. (2015). "Solving the JavaScript Problem". CodeNode (cit. on p. 5).

World Wide Web Consortium (2015). *W3C DOM.* URL: https://www.w3.org/TR/dom/ (visited on 12/07/2016) (cit. on p. 23).

Zakai, A. (2013). *Emscripten: An LLVM-to-JavaScript Compiler.* URL: https://github.com/kripken/emscripten/blob/master/docs/paper.pdf (visited on 14/07/2016) (cit. on pp. 2, 6).