Fall 2017

Multi Language Browser Support

Swapnil Mohan Patil
San Jose State University

Multi Language Browser Support

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Swapnil Mohan Patil

May 2017

The Designated Project Committee Approves the Project Titled

Multi Language Browser Support

by

Swapnil Mohan Patil

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2017

Dr. Thomas Austin	Department of Computer Science
Dr. Katerina Potika	Department of Computer Science
Dr. Robert Chun	Department of Computer Science

ABSTRACT

Multi Language Browser Support

by Swapnil Mohan Patil

Web browsers have become an increasingly appealing platform for application developers. Browsers make it relatively easy to deliver cross-platform applications. Web browsers have become a de facto universal operating system, and JavaScript its instruction set. Unfortunately, executing any other language than JavaScript in web browser is not usually possible. Previous approaches are either non-portable or demand extensive modifications for programs to work in the browser. Translation to JavaScript (JS) is one option but that can be challenging if the language is sufficiently different from JS. Also, debugging translated applications can be difficult.

This paper presents how languages like Scheme and Lua can be implemented in the web browser and shows how the web browsers can be extended to support multiple languages that can run in the browser simultaneously, interacting with each other seamlessly. In so doing, we hope to offer developers greater choice in languages for client-side programming.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Thomas Austin, who expertly guided me through my graduate education and my masters project. His constant mentorship, advice and support helped me to move in a right direction towards completion of the project. I would like to thank him for his time, help and efforts towards me and this project.

My deep gratitude also goes to Dr. Katerina Potika and Dr. Robert Chun for being on my defense committee. I would like to thank them for their time and efforts. Lastly, I would like to thank my friends and family. They supported and helped me to survive this stress and not letting me give up.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1

Introduction

The world wide web (WWW) got its popularity from its support for documents, images, videos, 3D graphics, and so on. The web is also excellent source of information that is made available through browsers.

A drawback of a basic web page is its limited behavior for dynamic content. This can be addressed by two types of programming extensions - One is server-side programming languages and other is client-side programming languages [2].

Server-side programming languages work on the server, which is situated across the network, and the browser must keep sending information to the server to process the input. There are lots of language choices available when it comes to programming on the server, such as Java, Python, JavaScript, and so on [2].

Client-side programming languages run in the browser, which helps to generate more dynamic content than simple web page can render [3]. But, there are very limited alternatives available when it comes to programming web pages in the browser such as JavaScript or Java.

As processing power in client-side machines is increasing, client-side programming is getting more popular among the software developer community. With that, feature list to be included in the specific preferences of client-side languages is growing.

But, currently JavaScript is the ubiquitous language that runs in all browsers. JavaScript is used to enhance user interactivity within web pages and can change the way a web page looks and acts at any time, based on user input.

In this paper we are going to address building multi-language support for browsers, where more than one language can work in the browser at the same time and all these languages will interact together to render the web page.

This paper is structured as follows :

Chapter 2 discusses the background research done in the field of multi-language environment; Chapter 3 discusses the Scheme environment and how it is implemented; Chapter 4 discusses the same details for Lua environment; Chapter 5 explains about how different environment interact with each other; Chapter 6 describes a sample how application can be built using Scheme, Lua and JavaScript working together; finally, chapter 7 concludes and presents opportunities for future work.

1.1 Multi-language Browser Environment

Every programming language is different; some are more concise than others, while some are better in execution speed, or are closely related to underlying system and hardware. A multi-language environment can support interaction between software modules written in different programming languages.

Nowadays, most of non-trivial systems are not written in only one language. Instead, many different languages are used; out of these are some general purpose programming languages like Ruby, Java, or JavaScript, and also some domain specific language (DSLs). A recent survey about open-source projects confirms that the use of multi-language environment is rather universal. So, multi-language environment is common among open source groups [4].

There are several benefits of having multi-language support in a browser, such as increase in productivity, benefit from multi-disciplinary client-side web pages development, and so on. Multi-language environment also supports different languages to work together, so that we can get best out of both worlds [5].

Currently, browser does not support multi-language environment. In this project, we built a multi-language environment in the browser for languages like Scheme [6], Lua [7], and JavaScript. We enable support for these languages in the browser, by building our own parser and interpreter for a small subset of languages like Scheme,

and Lua using JavaScript. These languages will interact with each other to render web page. We will see syntax and interaction of these languages in the upcoming sections.

1.1.1 Scheme Overview

Scheme [6] is a general purpose, simple, but powerful programming language. Scheme is widely used in computation research and education, it also is used in industrial applications like user interface designs, web navigators to virtual reality engines [8]. Scheme is formally standardized by IEEE [9].

Scheme program is block scoped. Variables and keywords in the Scheme program are lexically scoped. Occurence of the same identifier outside the block of code, refer to different identifier, otherwise reference is invalid. Blocks can also be nested in each other.

Scheme procedures are call-by-value. Procedures are also first object just like numbers, strings, and variable. Just like any other language, procedures can also be nested, and recursive. The same procedure can call itself.

Let's take a look into small overview of Scheme, which will give us an idea of getting started with writing Scheme programs.

1.1.1.1 Scheme Syntax

This section gives a small overview of Scheme as a language, to help us get some idea about Scheme.

Just like Lisp, Scheme programs are written as prefix expressions within parentheses for grouping. In Scheme, name of the operation comes before its operand [10].

Scheme program is combination of variables, objects, keywords, structured forms, comments, white spaces, and constant strings (numbers, vectors, strings, etc.) [11].

In C or most other languages, a procedure call to 'foo' with arguments 'baz' and 'bar' looks like:

```
foo(baz, bar);
```

But, in Scheme it is written as:

```
(foo baz bar)
```

Variable can be defined using 'define':

```
(define myvariable 5)
```

This will tell the Scheme to allocate variable 'myvariable' and assign value 5 to it. In the Scheme, variable value has to be defined always.

Procedure can also be created using 'define' as shown:

```
(define (two-times x)
        (* x x))
```

Above code creates a procedure with name 'two-times' with one argument.

Just like any other languages, if-else in Scheme can be implemented as follows:

```
(define (min a b)
        (if (< a b)
         a
         b)
)
```

It will create a procedure called 'min' with two arguments 'a' and 'b'. It will return minimum of both variables by comparing variables in if statements. If 'a' is less than 'b' then return 'a', otherwise return 'b'.

Scheme provides special procedure called lambda. Lambda does not give name to the procedure, it just returns the pointer to it.

We can use lambda in define statement, to assign it to variable as shown below:

```
(define double (lambda (x)
   (+ x x )))
```

Here, we are creating a procedure with one argument, and returning the pointer to it and we are storing that pointer into double variable using define.

1.1.2 Lua Overview

This section gives an overview of Lua programming language [7] and introduces the basic Lua concepts.

Lua is designed to be a small, simple, fast, and portable language, which can be easily embedded into other applications [12]. Lua is now popular among various types of application development such as robotics, web development, image processing, distributed systems, extensible text editors, and more [13]. Lua is one of the most popular scripting language for game development.

It is a procedural language with syntax like Pascal. It has control structures like (if , while, etc.). Procedure can have parameters, and local variables. Figure 1 below shows, implementation of factorial program in Lua.

```
function factorial (n)
  local i = 1
  local r = 1
    while i <=n do
      r = r * i
      i = i + 1
    end
  return r
end
```

Figure 1: Lua factorial example

1.2 Goal

Main goal of this project is to provide richer variety of languages for the client, by creating multi-language support for browser. It can be enabled by integrating client-side implementation of Scheme, Lua, with existing JavaScript. We will also dive deeper into questions like ‘‘How applications can benefit from it?” and ‘‘How can we make multi-language environment easy to use?” by actually creating a demo application using multi-language environment in browser.

Another goal is to create browser plugin and client-side libraries for parsing and interpreting, Scheme and Lua using JavaScript. Also, implement language subset of these languages to interact with the document object model (DOM).

This project does not try to achieve most efficient solution to multi-language browser support. but we think that our solution will definitely help the software community by showing a way of how languages can interact.

1.3 Approach

There are variety of possible approaches to built multi-language support for the browser. In this project, we are going to focus on approach of using the browser plugin (Mozilla and Chrome) for providing multi-language support for the browser.

1.3.1 Browser Plugin

Multi-language support will be integrated into the web browsers (Mozilla, Chrome) [14] using Mozilla and Chrome Plugin. The plugin will parse and then interpret any Scheme and Lua script on the page and will inject necessary events into the page again, as shown in Figure 2.

Whenever the new web page is loaded into the browser, plugin will scan the web page for any Scheme or Lua scripts on that page. Matching scripts will be parsed by the parser. Interpreter will interpret the parsed input into necessary DOM events.

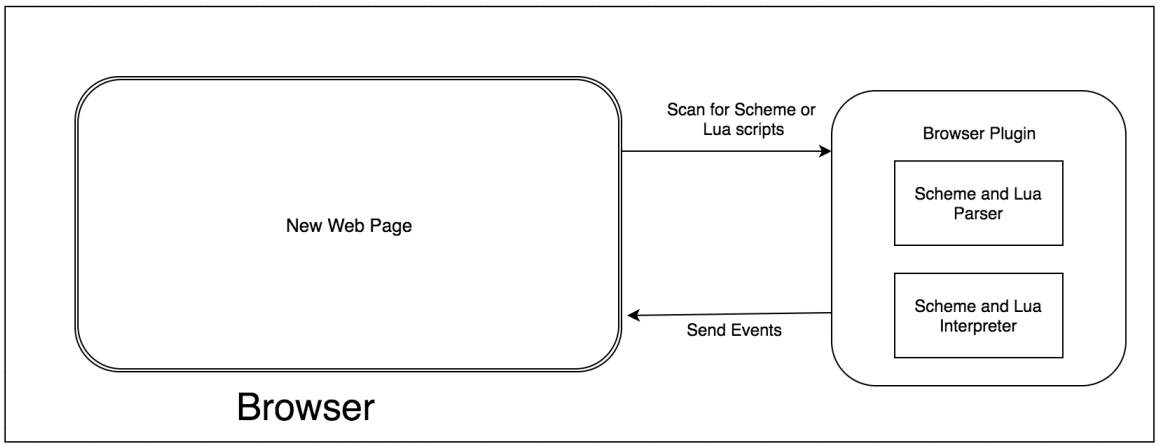

Figure 2: Browser Plugin Approach: Architecture

These events will be sent to the web page.

All the interaction between languages is managed by browser plugin. Advantage of this approach is, programmer does not have to worry about adding all the multi-language supporting libraries into the web page. Disadvantage of this approach is, plugin needs to be installed in the browser and some browsers are not yet supported by the plugin. Also, plugin needs a permission to read scripts from the web page. We overcome this disadvantage, by exposing Scheme and libraries, which can be included in the web page directly.

1.3.1.1 JS Library

If plugin is not available in the browser, multi-language support can be achieved by including supported libraries in the web page itself. Developer can include multi-language supporting libraries in the web page. After all libraries are loaded, browser will interpret different languages on the web page and help to achieve necessary interaction between these different languages. Libraries can be included in browser as shown in Figure 3.

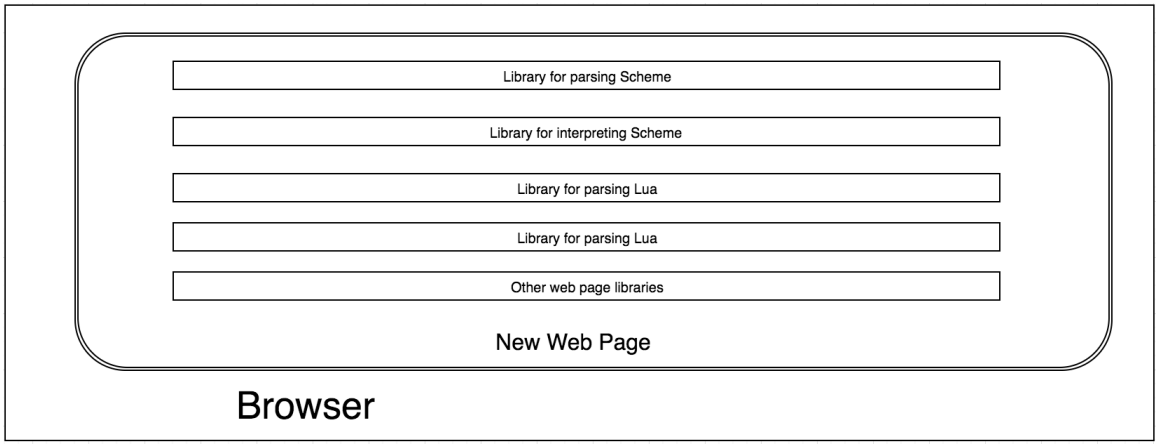

Figure 3: JS Library Approach: Architecture

1.4 Challenges

There are many challenges while building the multi-language support for any application. For building multi-language support for browser, these are some of the challenges that we came across.

1.4.1 Different APIs for DOM

To render web pages on the browser, language has to work with browser's DOM APIs. Every language has its own syntax for interacting with DOM APIs.

Currently, there is no implementation of Scheme, and Lua in the browser. For this project, we created our own subset of Scheme and Lua, which works with browser's DOM APIs, and JavaScript. We created our own parser and interpreter for these languages in browser, as discussed in Section 3 and Section 4

1.4.2 Interaction between different languages

Every developer uses different tools and technologies, each of which might have different types, features, and purpose. Each language has different way of representing types, data, object, and etc. So, it is historically very difficult to ensure language interoperability. For this project, we created basic utilities for language interaction to

happen smoothly, as discussed in Section 5..

1.5 Results

The contributions of this project;

1) Parser and Interpreter libraries for Scheme and Lua with support for DOM and interaction with JavaScript.

2) Browser plugin for Chrome and Firefox, which can execute Scheme and Lua scripts from any page opened in the browser.

3) A proof of concept Application, built using JavaScript, Scheme, and Lua working together.

CHAPTER 2

Related Work

In this section, we will have a look at what are the existing approaches and what efforts have been taken by others to allow languages to interact.

2.1 JVM/CLR Language Inter Operations

We first look into the Microsoft .NET's Common Language Runtime (CLR) and the Java Virtual Machine (JVM). Both these systems allow users to program in different languages allowing multi-language support.

2.1.1 CLR

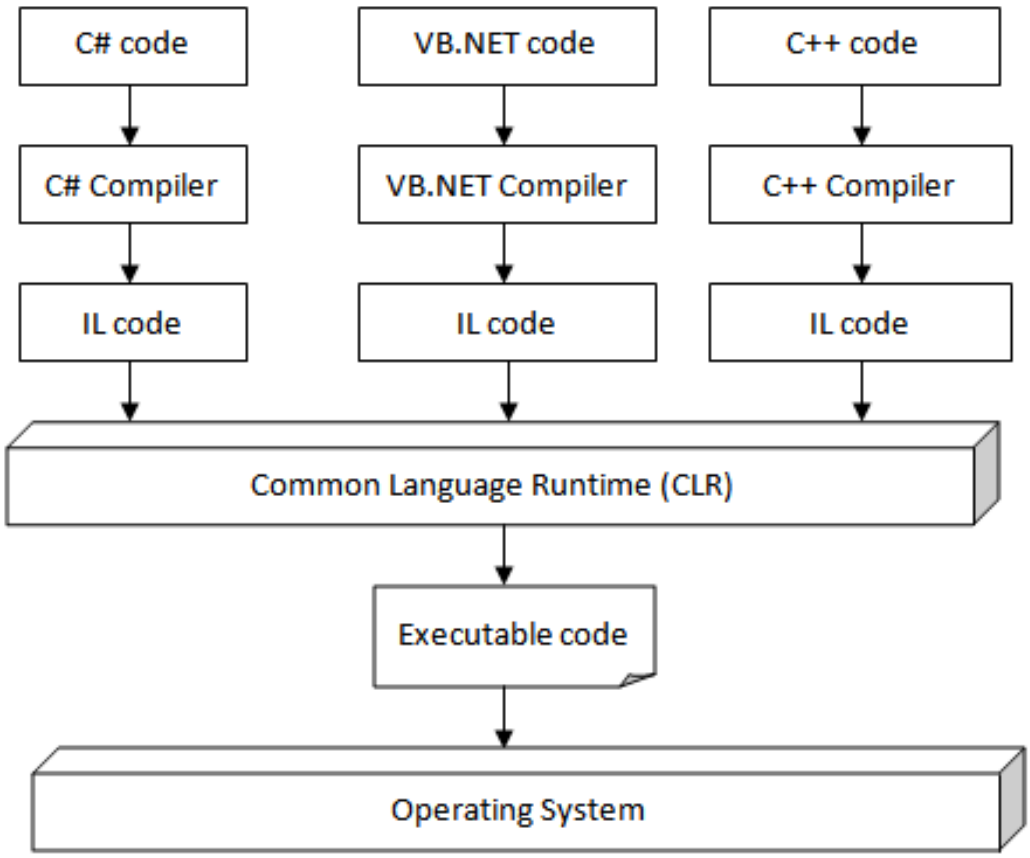

Figure 4: Common Language Runtime : Architecture [1]

The Common Language Runtime (CLR) is a virtual component of Microsoft's .NET framework. It provides runtime environment, which runs the code from various languages and also provides environment, which makes software development process very easy [15]. Figure 4 shows the architecture of CLR.

Tools and compilers expose the CLR's functionality, so that you can take advantage of this common language runtime's managed execution environment. The code written to target the common language runtime is known as managed code. This code can be benefited from features such as a cross-language integration, cross-language exception handling, versioning, and security by providing simplified way for component interactions [16].

The compiler converts the source code into the Intermediate Language Code (IL code). IL code is CPU independent instructions that can be converted into the native code. At the runtime, The CLR's Just In Time (JIT) compiler converts this IL code into native code.

Compilers and tools able to produce the output consumed by the CLR, because, format of metadata, type systems are defined by the public standard called the ECMA Common Language Infrastructure [17].

The Compiler emits metadata describing members, types, and references in your managed code to insure common language runtime provides services to managed code. This metadata is used to load and locate classes, load instances in memory, generate native code, enforce security [17].

Achieving even slight levels of language interoperability is pretty difficult because of the wide variation in programming language features, and implementations. The CLR makes it easy to build components, and application whose objects can interact with each other across the languages. Code written in different languages can be integrated, their behavior is tightly integrated. For example, the class written in one

language can be derived from the class written in other language and can call method of the class it is derived from. We can also pass instance of the class (Object) from one language to another. This is possible because language compilers, and tools uses common type system provided by CLR and they follow the CLRs standard rule for defining, creating, and pertaining types [15].

CLR provides following benefits:

1) Language inter-operation ability.

2) Performance enhancements.

3) Garbage Collection.

4) Support for exception handling.

2.1.2 JVM

The Java programming language [18] implementation compiles to Java virtual machine language (JVML) [19], also, known as Java-byte code. JVML is stored as java class files. It is either interpreted or compiled Just-In-Time (JIT) to native code by Java Virtual Machine (JVM). JIT allows efficient code as it is tailored as per the processor. Using such intermediate language provides many benefits [20]. Languages supported by JVM compiles to common byte-code, which helps in language interoperability.

Singer, discusses about difference between JVM and CLR [21].

2.2 Java Scripting APIs

After Java 6, Java supports incorporating code written in scripting languages directly in the Java. It enables developers to access their code written in scripting language directly in the Java applications. It began new generation of multi-language application called as polyglot applications (where the Java language can work together with other scripting languages). [22]

Developers were able to construct Java applications containing scripts developed in languages like JavaScript and Python. It uses JavaScript engine called Rhino [23] in Java 6, which is replaced by Nashorn [24] from Java 7. It is an implementation of the JavaScript engine, built entirely in the Java. It contains full support for the JavaScript [22].

This scripting functionality is provided by the javax.script package. The package contains very simple and small APIs for accessing JavaScripts. Scripting APIs are accessible through scriptEngineManager class. scriptEngineManager objects can search for script engines by means of jar file service detection mechanism.

According to the Java documentation [25] Way to access nashorn engine in the Java application:

1. Import the javax.script package.

2. Instantiate a ScriptEngineManager object.

JavaScripting APIs can be accessed through the ScriptEngineManager class. A ScriptEngineManager object instantiates ScriptEngine objects. It also maintains a global variable values shared by API.

3. Obtain instance of ScriptEngine from the manager with getEngineByName() method.

getEngineByName() method takes one string parameter with name of the script engine. To obtain instance of the nashorn engine pass nashorn. We can also use one of the following arguments "ecmascript", "ECMAScript", "Nashorn", "JavaScript", "javascript", "js", "JS".

After, we have the Nashorn engine instance, we can use to embed scripts in our Java application. Let's see some of the examples, which show the use of Nashorn:

Example 1 - Evaluating a statement

Figure 5 shows the example, which can evaluate "Hello world !" using Nashorn.

```
import javax.script.*;

public class EvaluateScript {
 public static void main(String[] arguments) throws Exception {
  ScriptEngineManager engineManager = new ScriptEngineManager();
  ScriptEngine eng = engineManager.getEngineByName(``nashorn'');

  // evaluate JavaScript code
  eng.eval(``print('Hello, World')'');
 }
}
```

Figure 5: Evaluating a statement: Nashorn

In the example shown in Figure 5, the script engine calls the eval() method and executes JavaScript string passed as an argument to eval() function.

Example 2 - Evaluating a JavaScript file

```
import javax.script.*;

public class EvalFile {
 public static void main(String[] arguments) throws Exception {
  ScriptEngineManager engineManager = new ScriptEngineManager();
  ScriptEngine eng = engineManager.getEngineByName(``nashorn'');

  eng.eval(new java.io.FileReader(``script.js''));
 }
}
```

Figure 6: Evaluating a JavaScript file: Nashorn

In the example shown in Figure 6, eval() method uses fileReader that reads JavaScript file and evaluates it.

Example 3 - Exposing a object from Java as a JavaScript's global variable

In the example shown in Figure 7, a file object in Java is passed to the Nashorn

```
import javax.script.*;
import java.io.*;

public class ScriptVars {
 public static void main(String[] arguments) throws Exception {
  ScriptEngineManager engineManager = new ScriptEngineManager();
  ScriptEngine eng = engineManager.getEngineByName(``nashorn'');

  // create File object
  File f = new File(``test.txt'');

  // expose File object as a global variable to the engine
  eng.put(``file'', f);

  // evaluate JavaScript code and access the variable
  eng.eval(``print(file.getAbsolutePath())'');
 }
}
```

Figure 7: Exposing a object from Java as a JavaScript's global variable: Nashorn

engine as a JavaScript's global variable. The put() method adds this file to global variable as the name file. Eval() function then calls Javascript code, which accesses this file variable as a JavaScript's global variable and executes the getAbsolutePath() method.

CHAPTER 3

Embedding Scheme in a browser

The Scheme development environment is the foundation for a number of fascinating applications in education and research. But executing Scheme program directly into the browser is not possible yet, since Scheme is not readily available in the browser. In such cases, JavaScript being de-facto of browser based applications, a JavaScript based and browser native implementation Scheme is desirable.

For this project we created Scheme in browser using JavaScript [14]. It provides supports for some of the core language features with additional feature of interacting with DOM to enable scheme to interact with the browser. The core of our environment is inspired by the scripting API [25] by taking the same approach to interact with other languages.

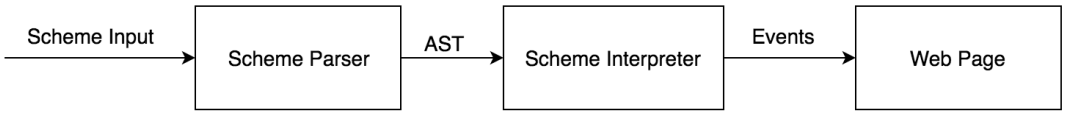

Figure 8: Scheme Environment using JavaScript: Architecture

As shown in Figure 8, Our Scheme library takes web page's embedded Scheme scripts as input. Our plugin will parse and then interpret the input scripts using JavaScript. This interpreted Scheme code then interacts with the browser, to help render web pages with more dynamic behaviour.

3.1 Interpreter

As discussed in previous chapters, browsers do not have a readily available support for the Scheme. To help enable the browser to understand code written in the

Scheme, we built a Scheme interpreter written in the JavaScript. Using our Scheme interpreter, browser will understand program written in the Scheme.

The Scheme code will be embedded into the web page using script tag as shown in the code Example 9,

```
<script type="text/scheme">
(
 (define square
  (lambda (n)
   (console-log (* n n)
  )
 )
)

(square 6)
)
</script>
```

Figure 9: Scheme script example

As shown in the code snippet in Figure 9, Scheme code can be added inside the webpage using "<script type="text/scheme" > </script>" script tag. Our interpreter supports multiple number of Scheme script tags.

In order for the browser to understand the Scheme, code snippet written inside Scheme script tags is parsed and interpreted by our browser plugin. In case, the browser plugin is unavailable then we need to include library reference of interpreter on the page, as shown in the example below,

```
<script src="simple-scheme-interpreter.js">
```

If the browser plugin is available, then this script file is added by the browser plugin, so we don't have to worry about adding it in the web page.

The interpreter consists of two parts:

3.1.1 Parser

After code snippets from script tags is read, first thing our library does is to parse the input code using PEG.JS and get the Abstract Syntax Tree (AST). The interpreter then uses this AST to interpret the code.

3.1.1.1 PEG.js

We used a PEG.js library to create a parser for the Scheme [26]. PEG.js is simply a parser written in the JavaScript. We can use it to create parsers for complex data and computer languages. A sample PEG.js grammar used to parse the Scheme code is shown in the code example 10,

```
var parser = PEG.buildParser(
start = multiexpression;\
validchar = [a-zA-Z_?!+\\-=@#$%^&*/.];\
spaces = \'' \''*;\
newline = [\\n]*;\
digit = [0-9];\
atom = spaces newline chars:validchar+ spaces
       newline   { return chars.join(\'\''); }\
       / spaces newline numbers:digit+ spaces newline
    { return parseInt(numbers.join(\'\'')); };\

list = spaces newline \''('' spaces newline
       expressionss:multiexpression+ newline spaces\'')\''
       spaces newline { return expressionss; };\
       expressions = spaces newline
       lists:list+ newline spaces { return lists };\
       multiexpression = atom / expressions ;');
```

Figure 10: Scheme PEG.js parser

Using the grammar shown in Figure 10, Scheme code is parsed into AST using a following statement.

```
var PegAST= parser.parse(s);
```

3.1.2 Interpreter

AST generated by the parser is interpreted by our interpreter written in the JavaScript, as shown in the code example below,

```
var pegRet = schemeInterpreter.interpret(PegAST);
```

The interpreted output is either DOM event or JavaScript code which the browser understands.

3.2 Scheme for Browser

Following the R5RS standard [27], variables are blocked scoped in the program. Blocks can be created with let, or define expressions, like:

```
(let ((x 10)
      (y 20))

(foo x y))
```

In the example shown above, variable x and y are blocked scoped. Both x and y are available to function foo.

Like the universal Scheme, our implementation of the Scheme supports following features:

- **Basic Types**
  - Integer
  - Real
  - Number
  - String
  - List
  - Char
- **Control structure**

  - Conditional
    * if then else cond case
  - Loop
    * do let(named let) dotimes
  - Assignment
  - eval

It also provides special language syntax to interact with the browser's DOM. We chose to include basic, most used browser functions for our Scheme DOM API. This is only available for browser.

First.

Second.

- **Browser Functions**
  - Dialog
    * (alert msg)

      Similar to, ‘‘window.alert” function.

    * (confirm msg)

      Similar to, ‘‘window.confirm” function. It returns a boolean value.

  - Event
    * (add-handler! selector event proc)

      Attaches an event handler to the specified selector. It returns the handler function, which handles the event.

    * (remove-handler! selector event js-handler)

      Removes the attached event handler from the specified selector.

      · eg (define h (add-handler! ”button1” ”click” (lambda (ev) ...)))

      · eg (remove-handler! ”button1” ”click” h)

- Element
  - (element-visible? elem)

    Returns the visibility of the specified element.
  - (element-toggle! elem)

    Toggles the visibility of the specified element.
  - (element-hide! elem)

    Hides the specified element.
  - (element-show! elem)

    Shows the specified element.
  - (element-remove! elem)

    Removes the specified element.
  - (element-update! elem html)

    Updates the specified element with the provided html.
  - (element-replace! elem x)

    Replaces the specified element with provided value x.
  - (element-insert! elem x)

    Appends the specified element with provided value x.
  - (element-select elem)

    Returns the specified element.

Apart from providing DOM APIs for the browser, our Scheme interpreter supports interface to some of the basic JavaScript functions as shown:

- **JavaScript language interface**
  - (js-eval str)

    Evaluate str as JavaScript code
  - Console

* (console-log obj1 ...)

  Similar to, JavaScript's ‘‘console.log”, it logs events to the console.

* (console-debug obj1 ...)

  Alias for ‘‘console-log’’

* (console-error obj1 ...)

  Logs the error message.

3.3 Approach

Our Scheme interpreter uses the browser plugin approach to help the browser to understand the Scheme code on the web page.

3.3.1 Browser Plugin

For this approach, the browser plugin (supported by Firefox, and Chrome) [14] will push the instance of PEG.js parser and our Scheme interpreter into every newly opened browser tab. In this way, user doesn't have to worry about adding the library script on the page. Our library will then parse and interpret all the code enclosed within Scheme script.

```
<script type=‘‘text/scheme’’>
(alert ‘‘Hello World’’)
(console-log ‘‘Hello World’’)
</script>
```

Figure 11: Scheme browser plugin example

As shown in Figure 11, we don't need to add any external libraries into our web page. Browser plugin will take care of interpreting the Scheme scripts. After executing the code in Figure 11 in the browser with our plugin installed, will alert user with ‘‘Hello World” as text.

3.3.1.1 JS Library

When the plugin is unavailable in the browser, the Scheme support is achieved in the web page by including Scheme interpreter and parser libraries in the web page itself. When the web page is loaded in the browser, libraries will read code snippet inside Scheme script and will interpret it, as shown in the Figure 12.

```
<script src=``./simple-scheme-interpreter.js''/>
<script type=``text/scheme''>
(alert ``Hello World'')
(console-log ``Hello World'')
</script>
```

Figure 12: Scheme without browser plugin example

As shown in code snippet in the Figure 12, instance of PEG.js and Scheme interpreter is embedded directly in the web page. Interpreter will interpret ``(alert "Hello World") and it will show an alert on the screen with ``Hello World" as text.

CHAPTER 4

Lua Environment

Currently, the web browser cannot execute the Lua code, as the Lua environment is not readily available in the browser. In this project, we will use the Lua VM provided by Mozilla [28] to provide Lua support for the browser. The Lua VM will create Lua environment where we will be able to execute our program written in Lua.

4.1 The Lua VM

The Lua VM runs in the browser by porting the entire ANSI C implementation of the Lua virtual machine to JavaScript using Emscripten [29] including garbage collection.

The Lua VM can be added in the web page by referencing ‘‘lua.vm.js’’ script on the browser, as shown below.

```
<script src=‘‘lua.vm.js’’></script>
```

Once the instance of the Lua VM is added on the web page, we can start writing our Lua code in the web page by including all the Lua code in the text/lua script as shown Figure 13.

```
<script type=‘‘text/lua’’>
...your lua code
</script>
```

Figure 13: Lua script example

The Lua VM interpretes the code written in ‘‘<script type=”text/lua” > </script>” tags, and it executes it as Lua code.

4.1.1 Examples

In this section, we will see some of the examples using the Lua VM in the browser.

1. Showing alert on the web page using Lua

Example shown in Figure 14 renders alert method of JavaScript global object using Lua.

```
<script src=''lua.vm.js''></script>

<script type=''text/lua''>
 js.global:alert('hello from Lua script tag in HTML!')
</script>
```

Figure 14: Showing alert on the web page using Lua

Executing the code from Figure 14 in the browser generates output similar to calling alert function from DOM, as shown in Figure 15.

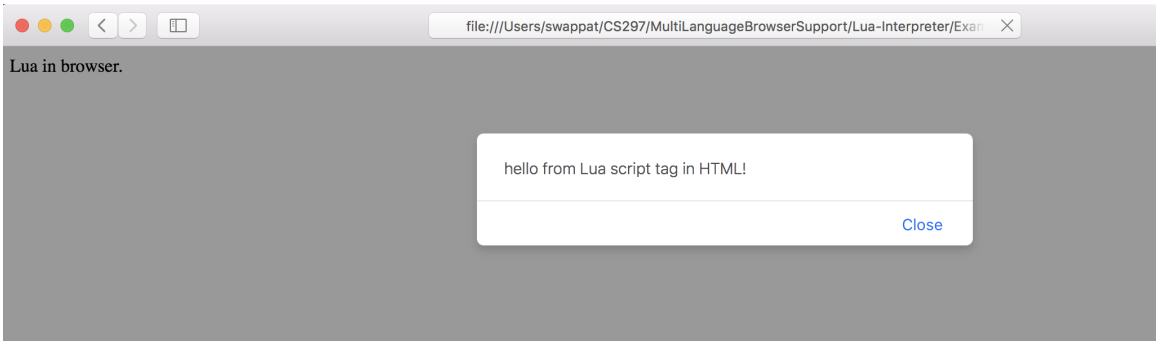

Figure 15: Showing alert on the web page using Lua: Output

2. Executing a Lua function in the browser

We can also define and call a Lua function in the web page, as shown in Figure 16.

Executing the code from Figure 16 in the browser creates a function called ''printName'' and assigns anonymous function to the local variable named ''sayHello''. After calling both functions, we see the desired output on the console, as shown in Figure 17.

```
<script src=''lua.vm.js''></script>
<script type=''text/lua''>
-- function
function printName (recipient)
print('Hello, '..recipient)
end
-- Anonymus function
local sayHello = function (recipient)
print('Hello, '..recipient)
end
sayHello('Swapnil')
printName('CS298 Project')
</script>
```

Figure 16: Executing a Lua function in the browser

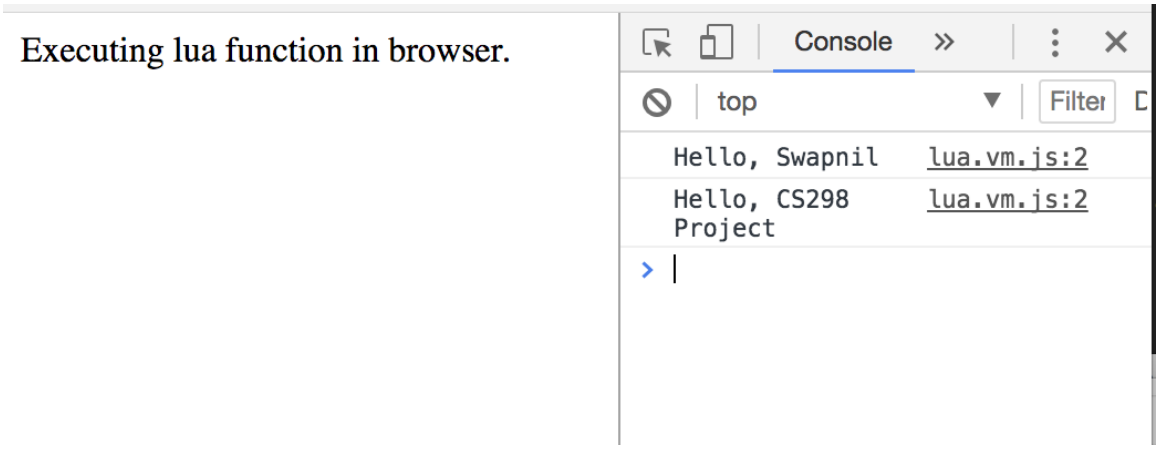

Figure 17: Executing Lua function in the browser: Output

4.2 Approach

Similar to the Scheme environment, we implemented browser plugin [14] approach to include Lua VM in the web page, to help browser understand Lua code on the web page.

For this approach, browser plugin (supported by Firefox and Chrome) will push the instance of Lua VM in every newly opened browser tab. In this way, user doesn't

have to worry about adding the library script on the page. Lua VM library will then interpret all the code enclosed within ‘‘type=text/lua” script.

In case of using a browser plugin, our example code to render alert in the browser is shown in Figure 18.

```
<script type=‘‘text/lua’’>
js.global:alert(‘hello from Lua script tag in HTML!’)
</script>
```

Figure 18: Lua alert with browser plugin

As shown in the code snippet from Figure 18, we don’t need to add any external libraries in our webpage to interpret Lua script, browser plugin will take care of it. Executing above code in the browser with our plugin installed will alert the user with ‘‘hello from Lua script tag in HTML!” as text.

4.2.1 JS Library

Whenever plugin is unavailable the Lua VM support is achieved in the web page by including the Lua VM library in the web page itself. When the web page is loaded in the browser, our library will read code snippet inside ‘‘type=text/lua” script and will interpret it, as shown in Figure 14.

CHAPTER 5

Interaction between Languages

After providing the Scheme and Lua support in the browser, we needed a way for these languages to interact with each other.

Every language is different and achieving even slight levels of language interoperability is pretty difficult because of the wide variation in programming language features and implementations. After going through all the challenges and possible solutions for this project, we decided to take an approach similar to the Java Scripting API [22] - code from other languages can be called by using the helper evaluate function provided by that language.

All languages will provide evaluate function, which will evaluate code native to that language and will return the results.

We will see the interaction between different languages with examples in the following sections.

5.1 Calling JS from Scheme

Our implementation of the Scheme provides and evaluate function called ‘‘js-eval”, which takes an argument of JavaScript code as string and it executes that JavaScript code from the Scheme environment, as shown in Figure 19.

```
<input type=``button'' id=``call''
value=``Click to call Java Script from Scheme'' />

<script type=``text/scheme''>
(
 (add-handler! ``#call'' ``click'' (lambda(ev)
 (js-eval ``var sayHello = function (tmp) {
  return 'Hello ' + tmp
  };''
 )
 (js-eval ``alert('Java Script Alert from Scheme - ``
  + sayHello('Swapnil'))''')))
 )
)
</script>
```

Figure 19: Calling JS from Scheme example

The code from Figure 19 adds a click handler to the input button with id ``call''. After clicking a button, it executes a js-eval function of Scheme, which creates JavaScript function called ``sayHello''. Next lines execute that JavaScript function by passing argument to it. Output generated by the code shown in the Figure 19, is shown in Figure 20.

Figure 20: Calling JS from Scheme example: Output

5.2 Calling Scheme from JS

Similar to calling JavaScript code from the Scheme, we can also execute the Scheme code from JavaScript. The web page gets the instance of our Scheme interpreter by calling evaluate method on that instance. JavaScript can execute Scheme code as shown in Figure 22.

```
<input type=``button'' onclick=``callScheme()''
id=``call'' value=``Execute Scheme Code
from Java Script'' />

<script>
function callScheme()
{
 var sayHello = scheme.evaluate(``(lambda (msg)
 (alert msg) )'');
 sayHello(``Calling Scheme method from JavaScript!!!!!'');
}
</script>
</html>
```

Figure 21: Calling Scheme from JS

In the code shown in Figure 22, we are evaluating function written in the Scheme and calling it from JavaScript. ``scheme.evaluate" function executes the Scheme code. In this case, we are storing Scheme lambda function in variable called ``sayHello'' and calling it with parameters from JavaScript, as shown in Figure 22

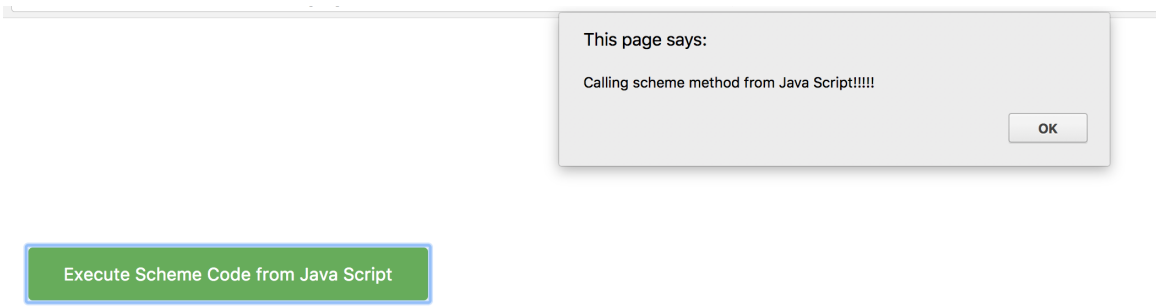

Figure 22: Calling Scheme from JS: Output

5.3 Calling JS from Lua

Lua interacts with the JavaScript using eval function on the JavaScript global object. Code snippet in Figure 23 shows how to write the Lua code in the Lua script and shows how to access JavaScript code from Lua script.

```
<script src=''lua.vm.js''></script>
<script type=''text/lua''>
-- global object in JS is the window
local window = js.global

-- Lua executing Java Script code
window:eval('function sayHello (message)' ..
 '{ alert(message); }' ..
 'sayHello(''Hello from Java Script calling from Lua'');'
)

-- Alert from Lua
window:alert(''hello from lua!'')

-- Accessing DOM APIS from Lua
local document = js.global.document
print(''This window has title '' .. document.title .. '''')

-- function
function printWindowSize ()
 local screen = js.global.screen
 print(''you haz '' .. (screen.width*screen.height) ..
 '' pixels'')
end

printWindowSize()
</script>
```

Figure 23: Calling JS from Lua

In the code snippet shown in Figure 23 the Lua environment is creating JavaScript function called ''sayHello'' and it is calling it using eval function on the JavaScript global object. Output of the code snippet is shown in Figures 24 - 26.

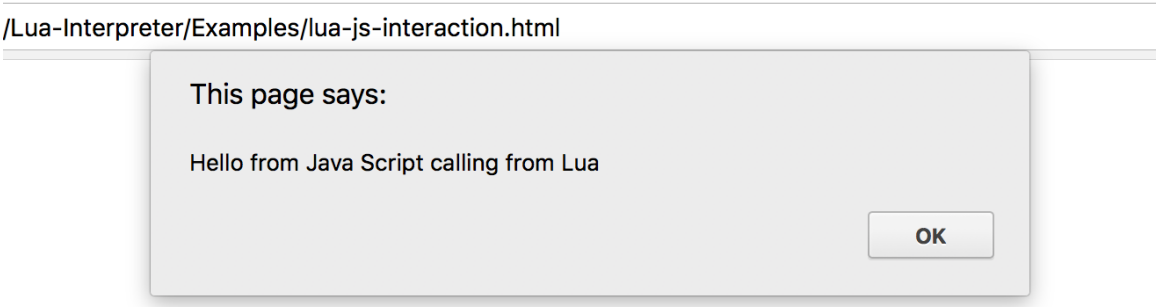

Figure 24: Executing JS from Lua Script: Output

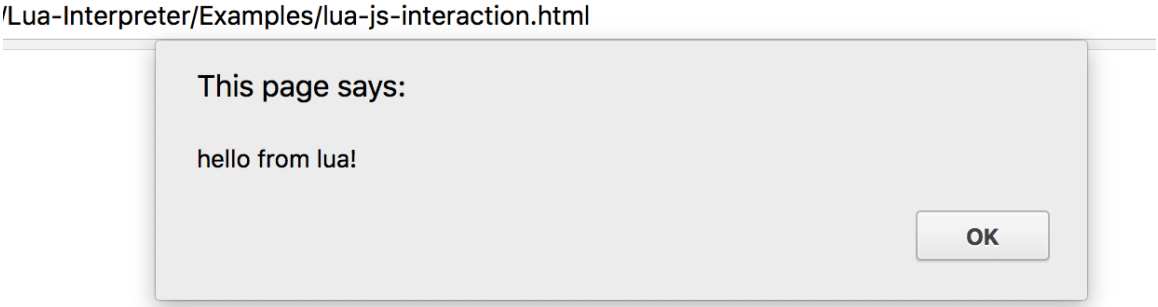

Figure 25: Alert from Lua : Output

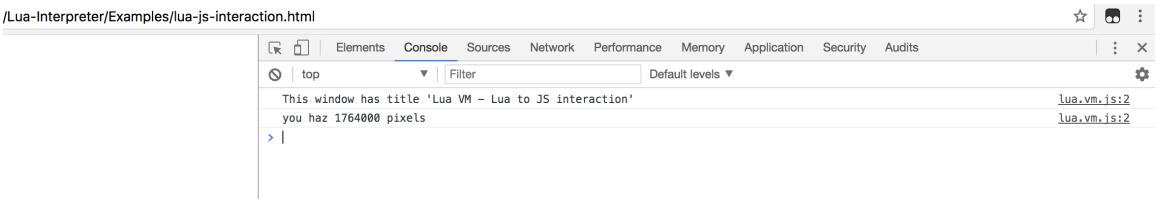

Figure 26: Accessing DOM from Lua: Output

5.4 Calling Lua from JS

The JavaScript interacts with the Lua using the ‘‘L.execute’’ function provided by the Lua VM. It accepts any Lua code as a string and executes it in JavaScript environment by calling the function, as shown in Figure 27.

```
<script src=``lua.vm.js''></script>
<script>
 function sayHello(message) {
  console.log(message);
 }

 L.execute( // Lua Function Declaration
  ``function printName (recipient) '' +
  ``print(`Hello, '..recipient)'' +
  ``end '' +

  // Call to Lua Function
  ``printName(`CS298 Project') '' +

 // Calling Java Script function from Lua
 ``js.global:sayHello(`Hello to JS function')'' +

 // Alert from Lua
 ``js.global:alert(`Hello from Lua') ''
);

</script>
```

Figure 27: Calling Lua from JS

The code snippet shown in Figure 27 calls the Lua code from JavaScript environment. It creates Lua function called ``printName'', which prints the message on the console. This function is called with a parameter ``CS 298 Project". It also calls JavaScript function called ``sayHello'' from the Lua code. And similarly, it alerts ``Hello from Lua" message using Scheme alert function. Output of the above code snippet is shown in Figure 28.

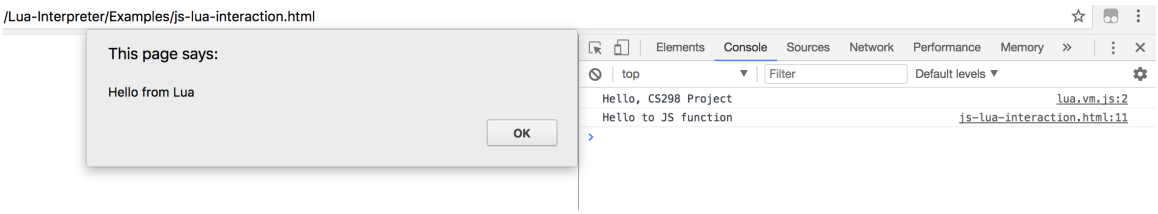

Figure 28: Calling Lua from JS: Output

CHAPTER 6

Multi Language App

After building support for various languages and their interaction in previous sections, this section demonstrates how to build a real world application using multiple languages. For this project we created hotel search application, which takes advantages of multi-language environment to render the app. This application helps the user to search for hotels by specifying the city and country.

Apart from taking advantage of multi-language environment (Scheme, Lua, and JavaScript), app also uses libraries like ‘‘Google maps APIs” [30], ‘‘Jquery” [31] etc. Home screen of the application is shown in Figure 29.

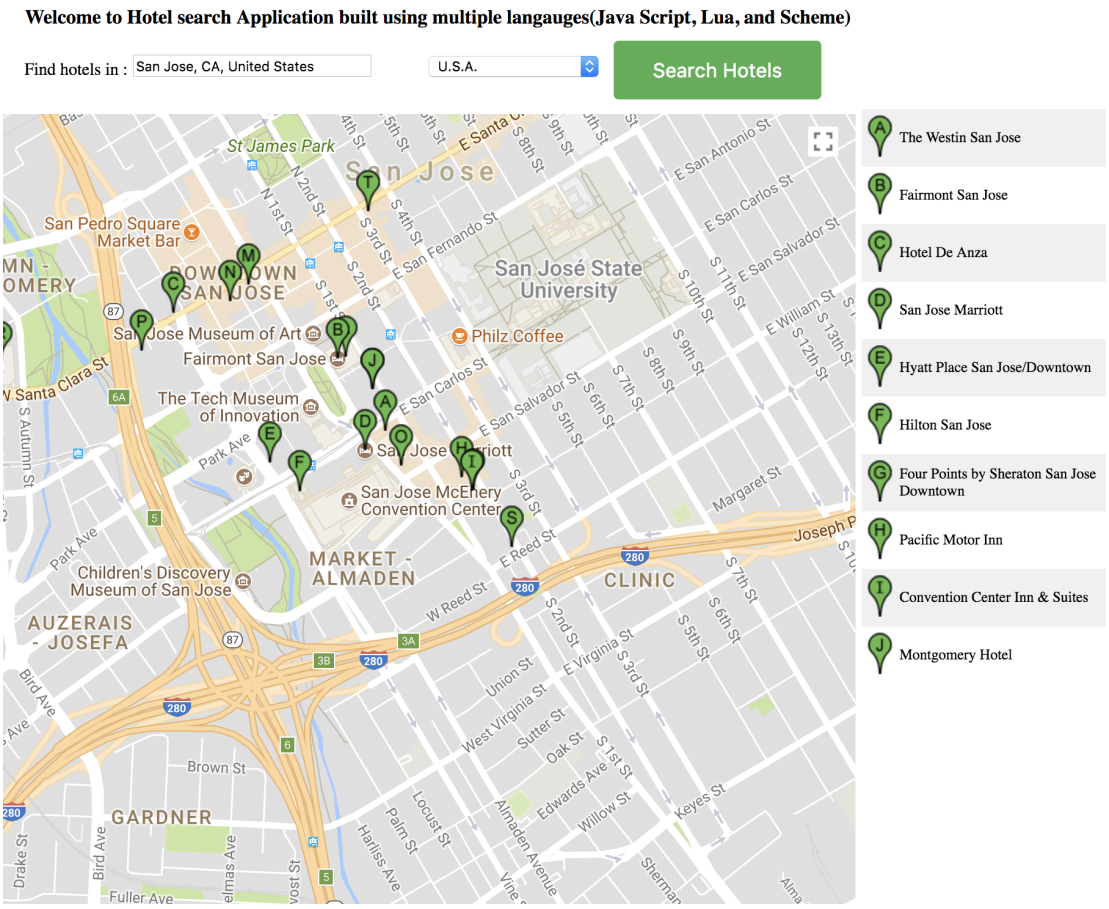

Figure 29: Hotel Search: Multi-language Application

Application provides the following functionalities :

- **Cities Auto Complete** : It provides the interface to enter the city.
- **Country Drop Down** : It provides the drop down to select the countries from a list of countries.
- **Search Button** : Button to search hotels in a selected city.
- **Map** : Provides a window to display google map on the screen.
- **Result** : Provides a content holder to show the search results.

JavaScript in the application handles rendering of the map and fetching search result from google places API. JavaScript maintains the reference to maps, countries

selected, search text in its global scope. JavaScript also calls Scheme's ‘‘getElem” function to get the reference of particular element on the page, as shown in Figure 30,

```
// Gets the map element reference.
var elem =scheme.evaluate(‘(getelem ”#map”)’);

// Gets the Result window reference.
var infocontent = scheme.evaluate(‘(getelem ‘‘#info-content’’)’);

// Gets the Search City text box reference.
var autocomplet = scheme.evaluate(‘(getelem ’‘‘#autocomplete’’)’);
```

Figure 30: JavaScript code

Scheme code in the application is responsible for initializing the app and provide on click functionality for the ‘‘Search Hotels” button, as shown in Figure 31.

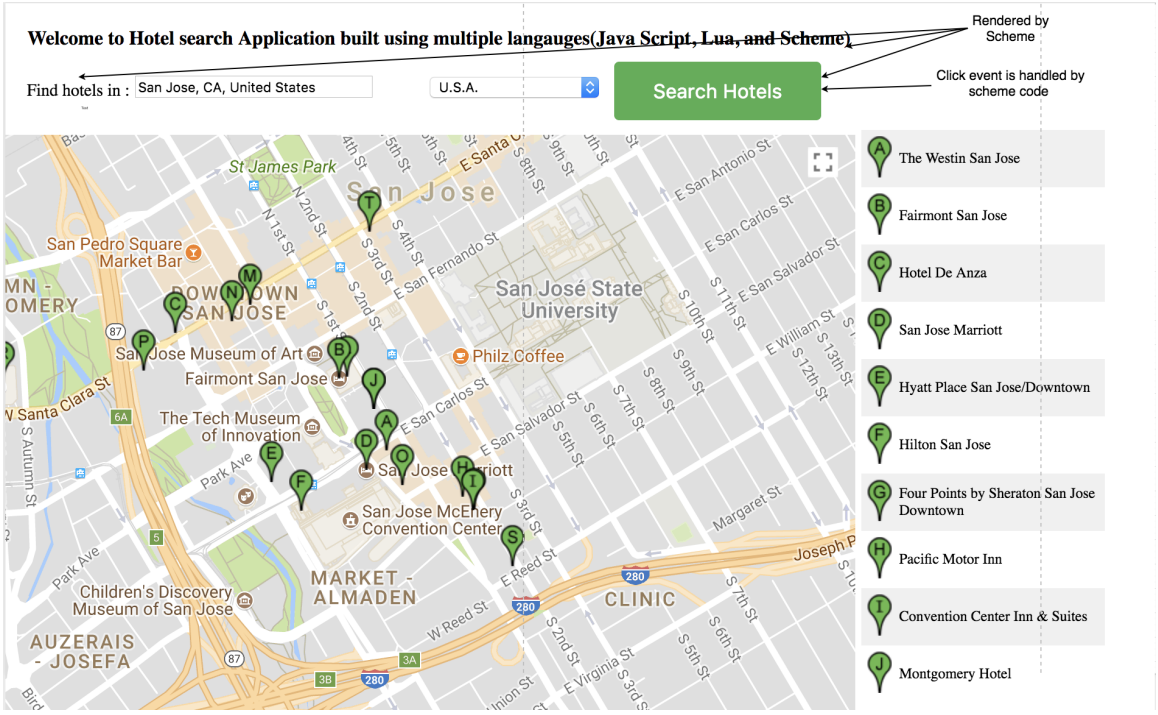

Figure 31: Scheme Contribution: Hotel Search Application

Code in Figure 32 shows the Scheme code which enables the functionalities shown in Figure 31.

```
<script type="text/scheme">
(
;;; Logs welcome message
(console-log "Welcome !!!")

;;; Declare InitialiseApplication function
;;; which initialisation the app
(define initialiseApplication
(lambda ()

;;; Updates the Header text to specified string
(element-update! "#header" "Welcome to Hotel search
Application built using multiple langauges
(Java Script, Lua, and Scheme)")

;;; Updates the Search Button text to specified string
(element-update! "#searchButton" "Search Hotels")

;;; Updates the Fild hotels text to specified string
(element-update! "#findhotels" "Find hotels in :")
)
)

;;; Adds a click handler on "Search Hotels" Button
(add-handler! "#searchButton" "click" (lambda(ev)
(js-eval "search()")
)))

;;; Calls initialise "initialiseApplication" function
(initialiseApplication)
)

</script>
```

Figure 32: Scheme Code: Hotel Search Application

Lua handles the drop down event on countries drop down. Lua gets the reference

to document object from JS global scope, it then sets the change event on drop down. When the country's drop down selection is changed, lua sets the focus of the map on the selected country, sets the zoom level and also calls the ‘‘clearSearchResult" and ‘‘clearMarkers" from JavaScript, to clear any result from previous search and markers on the map.

Lua code that handles these interactions is shown in Figure 33.

The complete code for this application is available in github [14].

```
<script type="text/lua">
--Gets the document reference from js global
local document = js.global.document

-- Function
function setAutocompleteCountry()

-- Gets the country drop down value
local country = document:getElementById('country').value

-- Gets auto complete refence
local autocomplete = js.global.autocomplete

-- Gets reference to map
local map = js.global.map

-- Gets refence to countries object in js global object
local countries = js.global.countries

-- Set the focus on the selected country
if(country == 'all') then
autocomplete:setComponentRestrictions()
map:setCenter()
map:setZoom(2)
else
autocomplete:setComponentRestrictions()
map:setCenter(countries[country].center)
map:setZoom(countries[country].zoom)
end

-- Calls Java Script clearResult function
-- to clear result results
js.global:clearResults()

-- Calls clearMarkers function from JS,
--  to clear markers oin the screen.
js.global:clearMarkers()
end

-- Adds the change event to country drop down.
document:getElementById('country'):addEventListener(
'change', setAutocompleteCountry);
</script>
```

Figure 33: Lua Code: Hotel Search Application

CHAPTER 7

Conclusion and Future Work

In this project, we implemented multi-language environment support for the browser. It gives developers flexibility in programming web pages in various different languages. An increased demand for computation in browser and availability of various different languages in the market makes it critical to have support for multi-language support for the browser. The libraries designed as a part of this project will help the developer program in languages like Scheme and Lua.

Although, having multiple languages at the time has its own complexities, there are various new research opportunities possible.

As a future work, we would like to expand these libraries to have complete functionality to access DOM APIs. We would like to explore the approach similar to CLR/JVM.

LIST OF REFERENCES

[1] ''Architecture of .NET,'' http://visualbasic-dotnet-hindi.blogspot.com/2012/07/architecture-of-net.html, accessed: 2017-11-09.

[2] S. WaiLok and A. Davison, ''Logicweb: Enhancing the web with logic programming,'' *The Journal of Logic Programming*, vol. 36, pp. 195–240, 1998.

[3] T. J. Hickey, ''Scheme-based web programming as a basis for a cs0 curriculum,'' *SIGCSE Bull.*, vol. 36, no. 1, pp. 353–357, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/1028174.971423

[4] P. Mayer, M. Kirsch, and M. A. Le, ''On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers,'' *Journal of Software Engineering Research and Development*, vol. 5, no. 1, p. 1, Apr 2017. [Online]. Available: https://doi.org/10.1186/s40411-017-0035-z

[5] J. Matthews and R. B. Findler, ''Operational semantics for multi-language programs,'' in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 3–10. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190220

[6] R. K. Dybvig, *The Scheme Programming Language, 4th Edition*, 4th ed. The MIT Press, 2009.

[7] R. Ierusalimschy, L. H. d. Figueiredo, and W. Celes, *Lua 5.1 Reference Manual*. Lua.Org, 2006.

[8] R. K. Dybvig, *The Scheme Programming Language: ANSI Scheme*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1996.

[9] ''IEEE Standard for the Scheme Programming Language,'' *IEEE Std 1178-1990*, 1991.

[10] S. Krishnamurthi, ''An introduction to scheme,'' *Crossroads*, vol. 1, no. 2, pp. 19–27, Dec. 1994. [Online]. Available: http://doi.acm.org/10.1145/197149.197166

[11] ''Scheme Programming Language description,'' https://www.scheme.com/tspl3/intro.html, accessed: 2017-11-28.

[12] W. C. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, ''Lua: an extensible extension language,'' *Software: Practice Experience*, 1996.

[13] ‘‘Lua projects.’’ http://www.lua.org/uses.html, accessed: 2017-11-09.

[14] ‘‘Multi language browser support,’’ https://github.com/swapnilpatil427/MultiLanguageBrowserSupport, accessed: 2017-11-21.

[15] A. Kennedy and D. Syme, ‘‘Design and implementation of generics for the .net common language runtime,’’ *SIGPLAN Not.*, vol. 36, no. 5, pp. 1–12, May 2001. [Online]. Available: http://doi.acm.org/10.1145/381694.378797

[16] ‘‘Common Language Runtime (CLR),’’ https://docs.microsoft.com/en-us/dotnet/standard/clr, accessed: 2017-11-29.

[17] J. J. Gough and K. J. Gough, *Compiling for the .Net Common Language Runtime*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[18] J. Gosling, B. Joy, and G. L. Steele, *The Java Language Specification*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[19] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[20] E. Meijer and J. Miller., ‘‘Technical Overview of Common Language Runtime,’’ http://www.csc.lsu.edu/~gb/csc7700/Reading/CLR.pdf, accessed: 2017-11-09.

[21] J. Singer, ‘‘JVM Versus CLR: A comparative study,’’ in *Proceedings of the 2Nd International Conference on Principles and Practice of Programming in Java*, ser. PPPJ ’03. New York, NY, USA: Computer Science Press, Inc., 2003, pp. 167–169. [Online]. Available: http://dl.acm.org/citation.cfm?id=957289.957341

[22] J. Juneau, *Nashorn and Scripting*. Berkeley, CA: Apress, 2017, pp. 529–551. [Online]. Available: https://doi.org/10.1007/978-1-4842-1976-8_18

[23] ‘‘Rhino,’’ https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino, accessed: 2017-11-09.

[24] ‘‘Project Nashorn,’’ http://openjdk.java.net/projects/nashorn/, accessed: 2017-11-09.

[25] ‘‘The Java Scripting API,’’ https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html, accessed: 2017-11-29.

[26] ‘‘PEG.js : Parser generator for JavaScript,’’ https://pegjs.org/, accessed: 2017-11-09.

[27] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson, ‘‘Revised5 report on the algorithmic language scheme,’’ *SIGPLAN Not.*, vol. 33, no. 9, pp. 26–76, Sept. 1998. [Online]. Available: http://doi.acm.org/10.1145/290229.290234

[28] ‘‘The Lua VM on the Web,’’ https://daurnimator.github.io/lua.vm.js/lua.vm.js.html, accessed: 2017-11-09.

[29] ‘‘Emscripten,’’ http://kripken.github.io/emscripten-site/, accessed: 2017-11-09.

[30] ‘‘Google Maps API,’’ https://developers.google.com/maps/, accessed: 2017-11-09.

[31] ‘‘Jqeury,’’ https://jquery.com/, accessed: 2017-11-09.