University of Waterloo
Cheriton School of Computer Science
CS 798 - Scripting Languages Project Final Report

A Comparison of Python, JavaScript and Lua Scripting Language Features

Afiya Nusrat, Alexander Pokluda and Michael Wexler
April 4, 2014

Abstract

In this report, we study three different scripting languages–Python, JavaScript and Lua– and compare the similarities and differences of the features that they offer. Our comparison is based on the implementation of a simple word game called Letter Lizard. We have implemented Letter Lizard in each of the three languages using similar data structures and algorithms as much as possible while taking advantage of each language's idiomatic features where appropriate. We start with an overview of the design and implementation of the game in each language and then compare the features of each language using example code from each of our Letter Lizard implementations. We finish by noting which language features made certain aspects of the game easier to implement and by summarizing the language features that we found most useful. By comparing these three scripting languages through the implementation of a non-trivial program, we hope to gain a deeper understanding of, and appreciation for, scripting languages in general and the tasks for which they are best suited.

Table of Contents

List of Figures

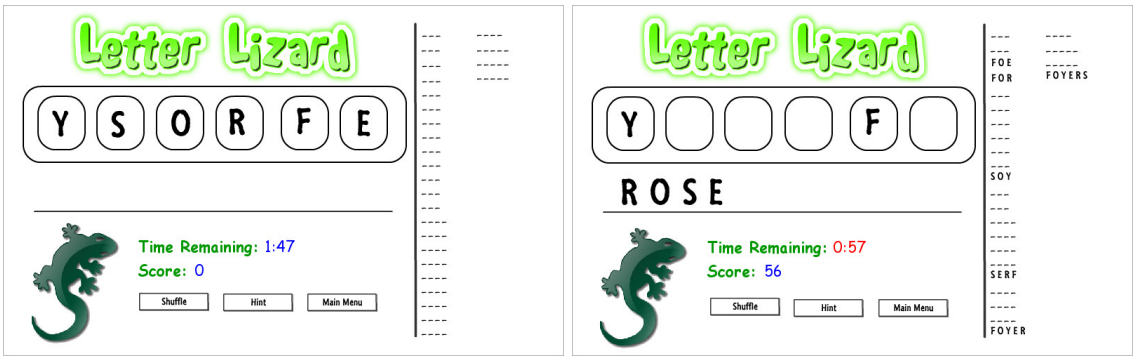
List of Listings

1 Introduction

Scripting languages is an increasingly popular and important category of programming languages. Programs written in scripting languages are often written for a special run-time environment that can be automated through scripts–shells and Web browsers are perhaps the two best known examples–or for a specialized domain, such as text processing. JavaScript and Lua are examples of the former. JavaScript is used to extend the functionality of Web pages displayed in a Web browser, while Lua is an extension language that is used in many commercial and free applications and is widely used in scripting video game engines. Lua is often chosen for this task because it is designed to be very fast and easy to embed. General-purpose scripting languages also exist and perhaps the most well-known one is Python. Python is a widely-used, general-purpose high-level programming language that emphasizes code readability whose syntax allows programmers to express concepts in fewer lines of code than would be possible in a language like C, enabling them to develop applications quicker. Scripting languages typically have a low barrier to entry and are easier for programers to get started with and provide a number of features that make them an important tool for increasing programmer productivity, and distinguish them from programming languages such as C, C++ and Java. For example, scripting languages are generally very high-level languages that provide a high level of abstraction. They are usually dynamically typed and provide automatic memory management. Many scripting languages can load and execute code dynamically in the context of the running program by passing a string consisting of program statements to an `exec()` function or something similar.

In this report, we study and compare three different scripting languages, Python, JavaScript, and Lua, and compare the similarities and differences of the features that they offer. Our comparison of these three languages is based on the implementation of a simple letter rearrangement game called Letter Lizard. In the game, the player is presented with a set of letters and their goal is to form as many dictionary words as possible from the set of letters before the timer runs out. In order to make the game more engaging and enjoyable, we implemented a number of features that enable the user to customize their gameplay experience by providing options to set the number of rounds, the time per round, and the level of difficulty for each game. We adhered to the same design for each implementation while also using the idiomatic features of each language as much as possible. The game proceeds as follows:

- On beginning the game, a welcome screen or a “Splash Screen” is displayed to the user. The user must hit the spacebar to proceed to the Main Menu



(a) *Main game screen mockup* **(b)** *Gameplay mockup*

Figure 1: *Two mockups from our design document demonstrating the proposed Letter Lizard game showing (a) the main game screen and (b) the gameplay.*

screen.

- The Main Menu screen allows the user to configure game options, such as the number of rounds, time per round and level of difficulty. After setting these options, the user clicks **Start** to begin a new game and is taken to the main Game screen.

- The Game screen displays the set of letters to the user and allows them to type words that can be formed from the letters. It also shows the round number, the amount of time remaining, the user's score, and placeholders for all of the dictionary words that can be formed from the set of letters. After typing a word, the user hits enter and the game engine checks to see that the word is a valid dictionary word, and, if so, reveals the word in the list of placeholders. The Game screen also has options to shuffle the set of letters, generate a hint for a word yet to be found, and return to the main menu.

All three implementations of Letter Lizard provide graphical user interfaces that are modelled after the mockups presented in our design document. The mockups showing the main game screen and gameplay are shown in Figure 1. Additionally, the internal data structures and algorithms representing the game state and gameplay are similar in all three implementations.

The set of letters and collection of dictionary words can can be formed from the set of letters is generated by a Python script called the *game generator* that can be run as a standalone, command line program or be imported as a module. The game generator works by loading a set of dictionaries, generating a set of scrambled letters, and then finding dictionary words that can be formed by the

set of scrambled letters. When run as a stand alone program, the default output of the tool is plain text where each line consists of the set of scrambled letters followed by all of the dictionary words that can be formed from those letters. Alternatively, the game generator can be provided with command line parameters that cause it to output the list of letters and words as JavaScript or Lua source code that initializes an object or table with the letters and words. The game generator accepts parameters that configure the number of scrambled letters in each game, the number of games to generate, and the difficulty of each game. The dictionary that we used (Spell Checking Oriented Word Lists by Kevin Atkinson) is partitioned into words that are more frequently used and less frequently used in common English. When choosing a harder difficulty, the list of words for the user to find will contain more words that are used less frequently. When generating the set of scrambled letters, the game generator picks each letter of the alphabet with a probability corresponding to the frequency that it appears in the dictionary (which we precomputed ahead of time). In order to find the dictionary words that can be formed, the game generator iterates through the dictionary and checks to see if each dictionary word can be formed from the set of scrambled letters, and if so, it adds it to the list of words for the set of scrambled letters.

The next three sections introduce each language and discuss the Letter Lizard implementation in that language. Section 2 compares the different features offered by each language using examples from our three implementations. Section 3 concludes by discussing which language features made certain aspects of the game easier to implement and summarizes the features that we found most useful.

1.1 PyLetterLizard: Python Letter Lizard Implementation

Python is a multipurpose, high-level programming language. Python has a friendly syntax, is easy-to-learn [1], and supports object-oriented, structured, and functional programming. It is used for a variety of applications by organizations such as Google, YouTube, NASA, and the New York Stock Exchange [4]. The friendly nature of Python makes it a very good tool to be used in an educational setting, and many find it useful as a first programming language to learn.

In the Python version of Letter Lizard (PyLetterLizard), we utilize three files, namely `letter_lizard.py`, `game.py`, and `config.py`. These files all work in tandem to allow PyLetterLizard to operate correctly. To run the application, one can type `python letter_lizard.py` at the command line in the PyLetterLizard directory in our Github repository (`uwaterloo-cs798scripting / group4`). PyLetterLizard requires Python 2.7 to be installed, as well as the corresponding version of Pygame. Upon launching, the function `main()` of `letter_lizard.py` is called. Before this occurs, however, the files `config.py` and `game.py` are included. `config.py`

declares configuration values that affect the placement of various game objects, as well as declares a couple of utility methods and objects. `game.py` declares a class Game which stores the state of a game in PyLetterLizard. We will discuss this module more in a moment.

When `main()` of `letter_lizard.py` executes, Pygame objects are instantiated, and some screen buttons are created. Then, the game enters an "infinite" loop, which does three things over and over: processes user input and events, updates the game state, and redraws the screen. We use the notion of a *game state* that is a value which reflects several states that the game can be in. This allows us to know when to draw objects that represent normal gameplay, as opposed to drawing the splash screen, menu screen, etc.

As mentioned previously, `game.py` contains the class Game, which stores the game state, and contains several methods which allow the game state to be altered. The reason we decided to use a class to represent a game is because it allows for the creation of new games quite easily, and separates the functionality of a Game into a discrete structure.

`game.py` contains several methods that `letter_lizard.py` utilizes to update the game state and process input and events. When a player types a letter, the method `process_letter` is called which checks to see if the letter exists in the set of letters displayed to the player, and updates the data structures correspondingly. `shuffle` is called when a user hits the space bar, which uses the `random.shuffle` function to randomly shuffle the puzzle word. This aids the player in finding new words in the puzzle. The method `draw` allows an instance of the class Game to draw itself onto the screen. The screen object is passed into the draw method as an argument. By having the game object be in charge of drawing itself, we avoid having to include any drawing functionality in `letter_lizard.py`, which gives the code tighter cohesion and looser coupling.

1.1.1 Constructs of Python used

```
1     def process_backspace(self):
2         self.message = ""
3         if (len(self.letters_guessed) >= 1):
4             letter_to_delete = self.letters_guessed[len(self.
    letters_guessed) - 1]
5             del self.letters_guessed[len(self.letters_guessed) - 1]
6             self.puzzle_letters_displayed[self.puzzle_letters_displayed
    .index('')] = letter_to_delete
```

Listing 1: *Basic constructs of Python used*

In the example above, we demonstrate several constructs of Python that we use in the Letter Lizard implementation. We demonstrate the ability for a Python class to define a member function (method), where the function has `self` as its argument. We use the `del` command in Python to delete the last element of the array `letters_guessed`. This syntax for deleting array elements is a bit different than in other programming languages.

```
1     def __find_length_counts(self, words):
2         word_lengths = [len(w) for w in words]
3         return {length:word_lengths.count(length) for length in set(
    word_lengths)}
```

Listing 2: *Demonstration of functional programming constructs in Python*

In the above example, we have a private method `__find_length_counts`, which when given an argument `words` (a list of words), will return a mapping from the unique counts of letters for each word to a count of how many words have that number of letters. For this method we utilize some functional aspects of Python. We use list comprehensions to iterate over all the lengths of `word_lengths`. We transform `word_lengths` into a set so that we only have the unique members. Then, we iterate over all the elements in that set, and for each one initialize an element of a new dictionary using a dictionary comprehension that maps a length value to the count of how many words have that length. The `dict()` construct transforms this list of tuples into a dictionary. The above method is used for generating placeholders for the words to be found in a game.

1.2 LetterLizardJS: JavaScript Letter Lizard Implementation

JavaScript is the "language of the Web" and has become an indispensable tool for Web developers. Client-side JavaScript scripts executed in Web browsers are able to bring life to Web pages by interacting with the user, controlling the behaviour of the browser, performing asynchronous communication with Web servers and altering the document that is displayed. Although originally introduced by Netscape for client-side scripting in 1995, it is increasingly more common for JavaScript to be used in other contexts as well. For example, a recent trend has been to implement server-side applications in JavaScript as has been demonstrated by the explosion in popularity of Node.js [5] and other server-side JavaScript frameworks. Not long after Netscape started shipping JavaScript in its Navigator browser, it submitted the language to Ecma International and it is now standardized as ECMA-262 and known as ECMAScript. There are several well-known implementations of the language that conform to the standard.

JavaScript is a dynamic, prototype-based object-oriented language with first-class functions. Much of its syntax has been influenced by C, C++ and Java, but

its semantics are actually very different. JavaScript borrows key design principles from Self and Scheme. JavaScript supports several different programming paradigms including object-oriented, imperative, and functional. JavaScript supports structured programming similar to C with many of the same flow-control statements such as `if`, `for`, `while`, etc. One notable difference, however, is the lack of block scoping for variables. Instead, JavaScript uses function scoping for variables, which means that all variable declared in a function are visible throughout the entire body of the function. JavaScript also contains a mechanism that tries to correct faulty programs by automatically inserting semicolons to complete statements, but quite often this masks more serious errors [2] or results in unexpected behaviour. We explore these issues further in Sections 2.2 and 2.1 respectively.

JavaScript has a dynamic type system in which types are associated with values, not variables. A few primitive types are provided by the language, such as Number, String, Boolean, null and undefined. Aside from the primitive types, everything else is an object. Objects are composite types that are comprised of properties: name-value pairs where the name is a String (or an integer for arrays, we will see more about this in Section 2.1.1) and the value is one of the primitive types or another object. Even functions are objects (with associated behaviour), which means that functions are first-class entities that may be assigned to variables and returned from other functions.

Each JavaScript function also contains a reference to the scope chain that was in effect when the function was defined, which is used to resolve variable names to values when the function is executed. A function, together with a reference to its scope chain is known as a *closure* (we will cover closures in Section 2.3.1). JavaScript usually runs in event-driven environments, such as the client-side environment of a Web browser, that make heavy use of closures and first-class functions for callbacks.

JavaScript supports object-oriented programming, but not in the classical sense. Rather than providing class-based inheritance, JavaScript provides *prototype-based inheritance*. We discuss object-oriented programming in Section 2.4.

Figure 2 shows eight screenshots from LetterLizardJS. When the player opens the Letter Lizard game in their browser, they are shown the Splash Screen (Figure 2a) that provides information about the game and prompted to press the space bar to continue. After pressing the space bar, the player is presented with the Main Menu (Figure 2b) that allows them to set the game options, namely the number of rounds, time per round and level of difficulty. Once the game starts, the player is presented with a set of letters represented as tiles on the game screen. The player can move the tiles to form words by typing on the keyboard (Figure 2d). At any time while playing the game, the player can get help to find additional words by

(a) *Splash Screen*

(b) *Main Menu*

(c) *Round Number message*

(d) *Game Screen*

(e) *Hint*

(f) *Good Job message*

(g) *Time's Up message*

(h) *Game Over message*

Figure 2: *Screenshots from the Letter Lizard JavaScript implementation showing (a) the splash screen, (b) the main menu, (c) a round number message that is displayed at the start of each round, (d) the main game screen as a user plays the game, (e) an in-game hint, (f) the "Good Job!" message that is displayed when a player finds all of the words, (g) the "Time's Up!" message that is displayed when the user does not find all of the words before the time runs out, and (h) the "Game Over!" message displayed at the end of the game.*

pressing the space bar or clicking the **Shuffle** button to shuffle the letters, or by requesting a hint by clicking the **Hint** button (Figure 2e). A number of in-game messages are displayed to the user while the game is being played, which are shown in Figures 2c and 2f-2h.

One of the main differences that we first noticed between Python, Lua and JavaScript had to do with the event-driven nature of the client-side scripting environment rather than the language itself: at the core of the the Python implementation is a game loop that processes events and redraws the screen. The Lua implementation does not have a game loop like Python, but it still has an update function that draws every frame of the game to the screen, whereas the JavaScript version does not have any such construct. Rather, the JavaScript version is entirely event driven. The game loop and update functions Python and Lua naturally led to a more procedural-based design, but the event-driven JavaScript environment naturally led to a more modular, object-oriented design. Most of the functionality of LetterLizardJS is implemented by the six classes shown in Figure 3. The Tile, Scrabmle, Builder and Word classes also draw themselves on the main game screen and their on-screen representation is shown in Figure 4. The Scramble class is responsible for creating Tiles for the letters that will be shown to the user and initially places those tiles on itself on the game screen. It also provides a shuffle method to rearrange the tiles when requested. The Builder class moves the Tiles to form words when the user types on the keyboard and also displays hints. The Game class is responsible for creating Word objects to represent words to be found. It also gets the characters that the user has typed from the Builder class and checks to see if they are valid words. If so, it causes the corresponding Word object to show itself in the word list. The Game class also updates the player's score when they find a word and manages the game timer.

The code for our JavaScript implementation can be found in our Github repository (`uwaterloo-cs798scripting / group4`) under the LetterLizardJS folder. To play the game, you simply need to load the `index.html` file in your browser; however, due to the security policies of most Web browsers that restrict the functionality of scripts loaded as local files, you will likely have to run an HTTP server on your local machine and load the `index.html` file through your server in order to play the game. Alternatively, we invite you to play the game using a server that we have set up at using the following URL: `http://dahu.in/static/LetterLizardJS/index.html`.

1.3 LuaLetterLizard: Lua Letter Lizard Implementation

Lua is a light weight programming language designed as a scripting language. Lua is generally described as a "multi-paradigm" programming language because

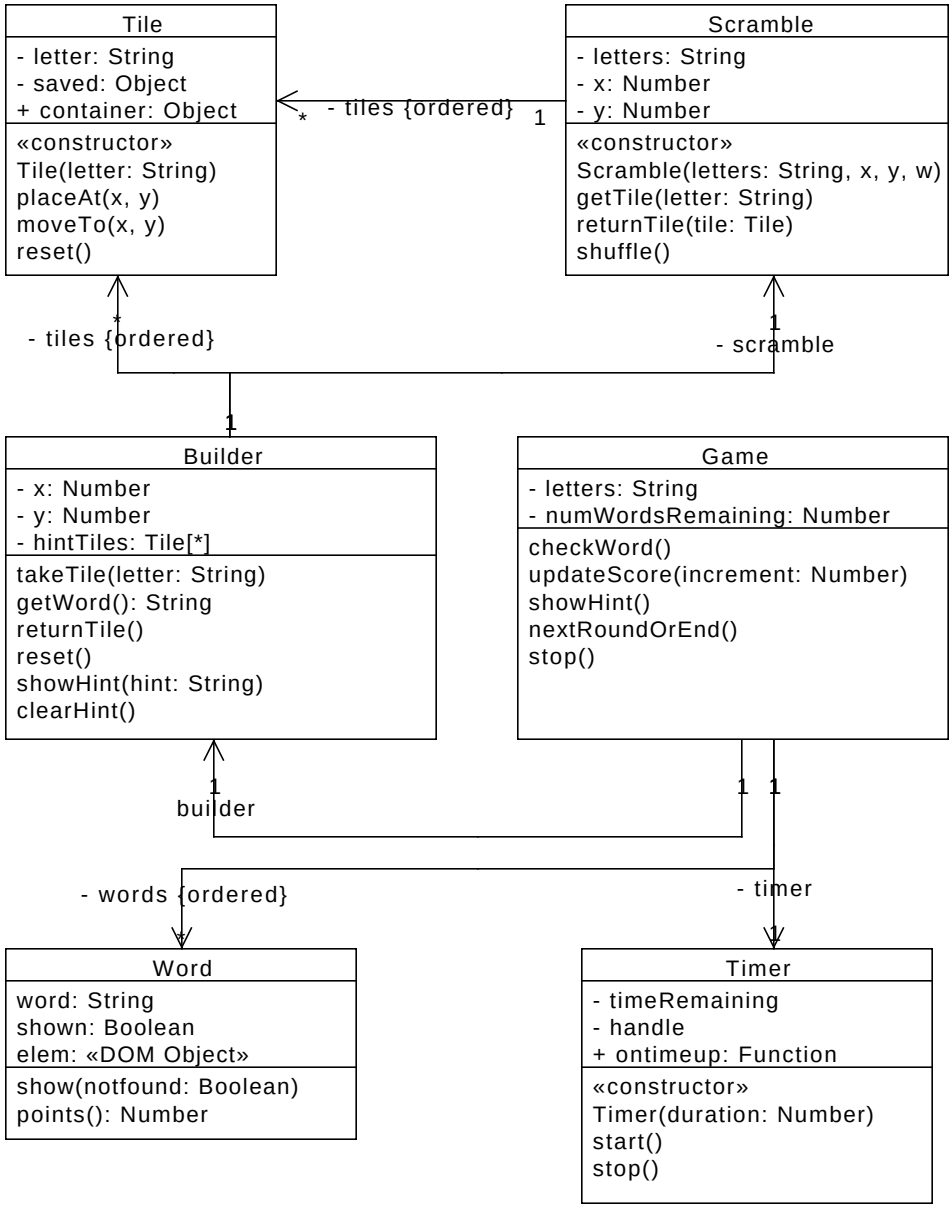


Figure 3: *A class diagram showing the classes that make up the Letter Lizard JavaScript implementation. The event-driven, callback-based client-side scripting environment naturally led to a modularized, class-based design for this implementation.*

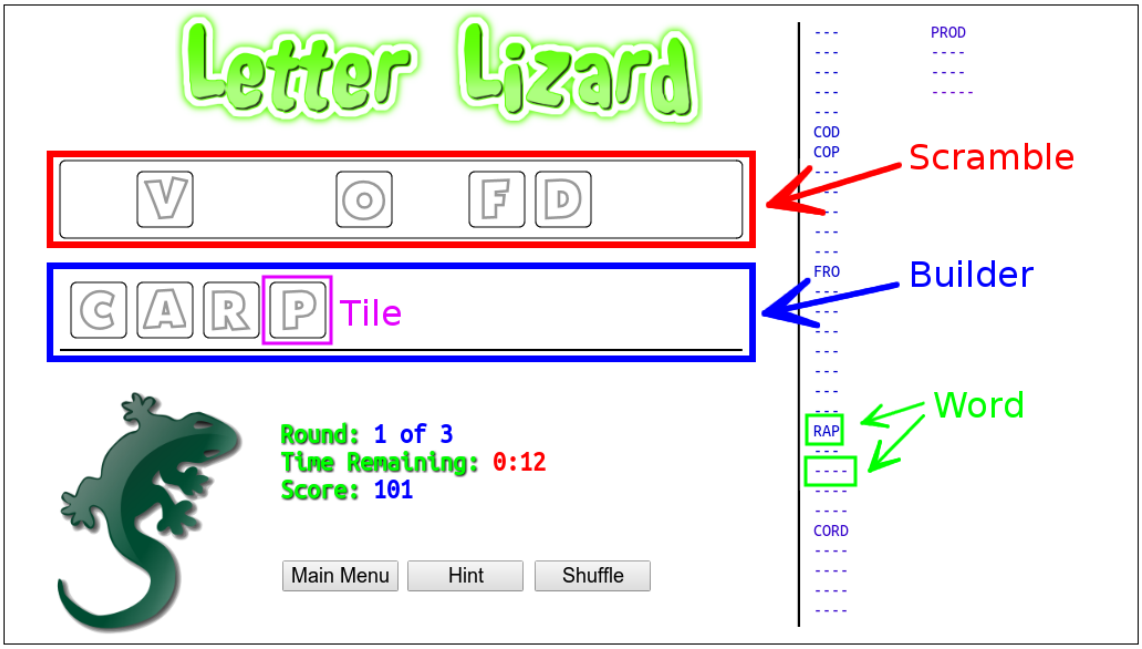


Figure 4: *The on-screen representation of the Scramble, Builder, Tile and Word classes.*

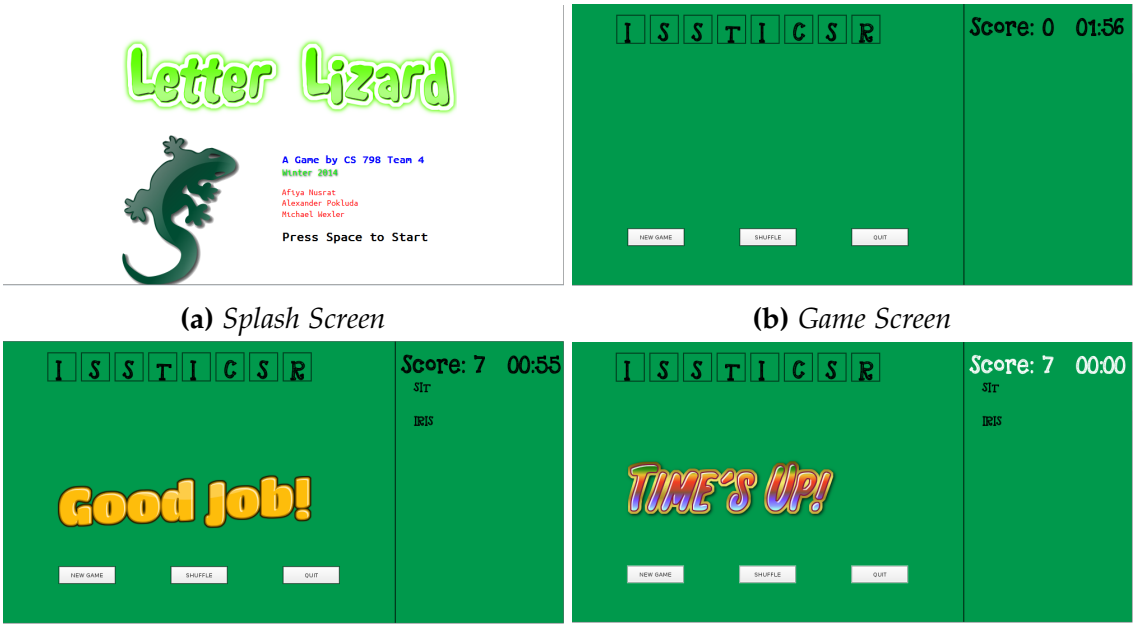
it provides a small set of extensible features that can be extended to fit various problems rather than providing definite features for a specific paradigms. Lua has a short learning curve, is easy to pick up, and has a number of features which endorse its reputation. It does not, however, support inheritance but allows it to be implemented by using metatables. A table is a fundamental data structure in Lua that can be used to represent everything. Tables are in fact the only data structure in Lua and can be used to implement other data structures such as arrays, sets, lists, records and other data structures. Lua tables can be used to implement features such as namespaces and classes, and in this sense has a lot of similarities with JavaScript. For instance, tables in Lua are quite similar to JavaScript objects– the only difference being that in JavaScript, objects can be indexed with only string or integer values. Lua tables, on the other hand, can be indexed with any value of the language (except nil). Lua is dynamically typed and is light enough to fit on any operating system. The Lua interpreter is extremely lightweight; it is only 180 kB when compiled, and therefore is very fast when compared to Python and JavaScript. It is a minimal but powerful language. By including only a minimal set of data types, Lua attempts to strike a balance between power and size [6]. Lua is often used in game development because of its speed, portability, embeddable nature and simple but powerful design. In order to enable embedding in other languages, Lua provides a well documented API to extend programs written in other languages.

A number of important language features in Lua include first-class functions, dynamic module loading, automatic coercion, and closures. Although Lua does not support the object-oriented concepts of classes and methods, they can be simulated using tables and first-class functions. By placing functions and related data into a table, objects are formed. Inheritance can be implemented using metatables. It is similar to JavaScript in this way as there is no explicit concept of a class in Lua; rather, prototypes are used similarly to JavaScript and new objects are created by a factory method, i.e, by cloning existing objects. Lua provides "syntactic sugar" to facilitate object-orientated programming techniques. In order to declare member functions inside a table, the programmer can use `table:func(args)`. The colon operator adds a hidden "self" parameter to function calls.

1.3.1 Lua Game Framework

We used a framework called LÖVE for developing the Lua-based implementation of Letter Lizard. LÖVE is a free, open source framework for building 2D games in Lua. The LÖVE framework has cross platform adaptability and was installed from the LÖVE homepage [3]. To run a game, LÖVE can load a game in two ways: from a folder that contains a `main.lua` file, or from a `.love` file that has a `main.lua` in the root directory. LÖVE utilizes callback functions to perform various tasks. LÖVE provides placeholders for callback functions in order to structure the game logic. For instance, the `love.load` function is called only once when the game is started and is usually used to to load resources, initialize variables and set specific settings. The `love.draw` function is where all the drawing happens and if any of the `love.graphics.draw` objects are called outside of this function, they will not have any effect. Figure 5 shows some screenshots from the Lua version of Letter Lizard.

The game structure consists of five different Lua modules. These are `main.lua`, `conf.lua`, `games.lua`, `helper_functions.lua`, `gamestate.lua` and `config.lua`. The `main.lua` module is the driver of the game and utilizes all the other modules. `conf.lua` contains game configurations and is run exactly once before `main.lua` by LÖVE. `games.lua` contains 300 games pre-generated using game generator program. `gamestate.lua` was taken from a small Lua utility library and enables us to maintain state information within the game and allow us to switch between states. Since Lua is a very low-level language, we had to code a lot of functionality on our own. These free standing functions are contained in the `helper_functions.lua` module. `config.lua` contains additional game configurations which are used by `main.lua` to drive the game. In order to execute the game we need to run the LÖVE executable and give the directory containing the



(a) *Splash Screen* **(b)** *Game Screen*

(c) *Gameplay Mode* **(d)** *Game Over*

Figure 5: *Four screenshots from LuaLetterLizard showing (a) the splash screen, (b) the main game screen, (c) the "Good Job!" message that is displayed after the player finds a word, and (d) the "Time's Up!" message that is displayed when the timer runs out.*

`main.lua` file as an argument. LuaLetterLizard can be launched by running the command `love LuaLetterLizard` in our `uwaterloo-cs798scripting / group4` Github repository directory.

1.3.2 Game Logic and Implementation

The `main.lua` file contains the main game logic structured within callback functions. In this file, we declare two Lua table structures, menu and game. We implement metatables a powerful feature provided by Lua, to enable the concept of state within the game. Within the menu table we declare functions which are associated with the menu and are called only when menu is the current state of the game. Within the game structure, we declare functions which are called only when the we enter the gameplay mode.

The `menu` table contains functions that are called when we start the game and the splash screen is loaded, as shown below. The function `menu:init()` is called when the game is first loaded into memory and the splash screen is displayed. In order to actually draw the splash screen, the function `menu:draw()` is called. Within the `love.keypressed()` function we check if the player has hit the space bar, and if so, the state of the game is switched to "game" and the associated callback functions are then called. All of the game play logic is structured within the `game` table. LÖVE is different from Pygame in that there is no infinite loop which keeps running. Rather, callbacks are called as and when required. However, there is still an update function, `love.update()`, that is called to render each frame of the game. It is passed a parameter `dt` which is the time passed since the last time update was called.

```
1 local menu = {}
2 function menu:init()
3     splash = love.graphics.newImage("splash.png")
4 end
5
6 function menu:draw()
7     love.graphics.setColor(255,255,255,255)
8     love.graphics.draw(splash, 0 ,0)
9 end
```

Listing 3: *LuaLetterLizard `menu:init()` function*

Once we enter into the game mode, the corresponding callback functions defined in the `game` table are called. For instance, in the following code snippet, the function `game:draw()` is called to draw objects onto the screen while we are currently in the gameplay state i.e., the default `love.draw()` function is overridden. Hence, callbacks defined when in a particular state are called as and when required.

```
function game:draw()
    love.graphics.setColor(black)
    love.graphics.line(700,0, 700, 500)
    for i, letter in ipairs(letters_guessed) do
        x = letters_guessed_left + i * square_width + i * spacing
        y = letters_guessed_top
        love.graphics.rectangle("line", x, y, square_width,
    square_width)
        love.graphics.setFont(number_fnt_40)
        love.graphics.print(letter, x + square_width/4, y +
    square_width/5)
    end
    ...
end
```

Listing 4: *LuaLetterLizard `game:draw()` callback method*

In order to maintain the game state, we utilized an external Lua library called "hump." This is a lightweight library which contains helper code for embedding a set of different functionality within our code. Since Lua is very low level, we had to code a lot of functionality ourselves, and using hump minimized extra work for us considering the time constraints. This was another advantage of using the LÖVE framework, as LÖVE has a very good online community; many helpful libraries have been implemented by other Lua game developers and can be used under the open source licenses. Another important part of the game logic is checking for events. For this, we utilized the callback functions defined by LÖVE, `love.keypressed()` and `love.mousepressed()`. Within the former callback we check for events that fire when a keyboard key is pressed by the player and define logic for the corresponding action that needs to be taken, whereas in the later we check for events when the player click a particular button which fires the corresponding actions. In the following code snippet, we check if the space key has been pressed by the player while we are in state "menu" and if so, we switch the game state to "game" to begin playing the game.

```
function menu:keypressed(key)
    if key == ' ' then
        Gamestate.switch(game)
    end
end
```

Listing 5: *Handling the keypressed event in LuaLetterLizard*

2 Scripting Language Feature Comparison

Now that we have introduced the three implementations of our Letter Lizard game, we will will compare and contrast the differences between each language, noting their strengths and weaknesses, using example code from our implementations where possible. When coding the Letter Lizard game in each language, we tried to follow the same structure as much as possible, however, as was noted in Section 1.2 the event-driven environment of client-side JavaScript naturally led to a different design for that implementation. Due to space constraints, we will skip over some of the differences that we noticed between the types, variables and values (including the numeric, boolean and string types and the null, undefined and nil special values) offered by each language. We will, however, discuss the differences between the data structures offered by Python (i.e. lists, tuples and dictionaries), JavaScript (i.e. objects and arrays) and Lua (i.e. tables and arrays). We noted differences between the syntax and semantics of expressions and statements in each language, as well as the flow control constructs offered, which we will omit; however, we will discuss variable scope, functions, and closures. The first major difference that we noticed between the languages was non-functional: each employed a different lexical structure, which we discuss next.

2.1 Lexical Structure

Since we implemented the game in three different languages we were able to examine the difference in lexical structure quite minutely. The syntax of a programming language defines the set of symbols that are considered to be a correctly structured program in that language. Python is designed to be a highly readable language and prefers the use of English words instead of punctuation marks. As compared to other languages, Python uses white space indention rather than curly braces or keywords to delimit blocks. This is a strength of the language as good indention is enforced by the language itself and makes for highly readable, clean code. However, on the downside, tabs and spaces can be easily mixed up leaving bugs in the code.

JavaScript uses blocks delimited by curly braces which means that code can be "minimized" for the Web. However, automatic semicolon insertion can lead to errors and is generally not considered very good by JavaScript programmers. This is one of the controversial syntactic features of JavaScript. For instance, consider the following code snippet:

```
1 a = b + c
2 (d + e).print()
```

Listing 6: *An example where automatic semicolon insertion in JavaScript may lead to unexpected results*

In this example, the code is not transformed by automatic semicolon insertion, because the parenthesized expression that begins the second line can be interpreted as an argument list for a function call:

```
1   a = b + c(d + e).print()
```

Listing 7: *The interpretation of the previous example*

In Lua the statements and blocks are delimited by newlines and keywords to begin and end constructs which helps to minimize errors, but it can lead to extremely messy code unless indention conventions are strictly followed.

2.1.1 Data Structures

A data structure is a particular way of storing and organizing data in a computer program so that it can be used efficiently. There are different kinds of data structures available which are suited for performing different tasks. In our implementations of Letter Lizard, we extensively used the different data structures provided by each language for holding, structuring and manipulating our data efficiently. All three of our implementations used data structures for holding the scrambled puzzle letters, letters guessed correctly, letters to be displayed on the screen and so on. We will now describe the data structures in all three of our implementations.

Python: Arrays, Lists, Tuples and Dictionaries Unlike JavaScript and Lua, which each provide only one data structure, Python provides a rich set of built-in data structures. The primary data structures used by Python programs are:

list mutable sequences of items of arbitrary type

tuple immutable sequence of items of arbitrary type

range immutable sequence commonly used for looping

set mutable container of items of arbitrary type

dictionary mutable mapping of keys to corresponding values

Python provides a concise way to create lists, sets and dictionaries called "comprehensions." These are commonly used to initialize new data structures where each element is the result of certain operations that are applied to each member of another sequence or iterable object, or meets some sort of filter condition. We previously gave an example in Section 1.1.1 where we had used set and dictionary comprehensions to initialize a set and dictionary. List comprehensions also facilitate functional programming in Python.

JavaScript: Objects JavaScript has one fundamental datatype: *object*. Everything that is not either a primitive type, null, or undefined is an object. (The primitive types in JavaScript are number, string, and boolean. Although they are not objects, they behave like immutable objects). Objects are composite values that aggregate multiple values, which may be primitive types or other objects, and allow you to store and retrieve those values by name. Objects are similar to Python's dictionary and Lua's table data structure, but unlike Python's dictionary, objects are a core feature of JavaScript. All variables are contained within objects and, as we will see later in Section 2.3, even the scope of a variable is implemented in terms of a chain (or linked list) of objects. Lua's tables are very similar to JavaScript objects and play a similar role in the implementation of the language, but they are even more powerful.

JavaScript objects are dynamic–rather than having a static type, properties can be added and removed at run-time and support "duck typing." LetterLizardJS makes use of many user-defined objects. The following code defines an object `letterpoints` that maps letters (property names) to the number of points that the letter is worth (property values) when included in a word found by the player:

```
1 var letterpoints = {
2   'A': 3,
3   'B': 8,
4   'C': 6,
5   ...
6   'Z': 10
7 };
```

Listing 8: *A user-defined object in JavaScript*

Another example of a user-defined object in LetterLizardJS is shown below. This code snippet is from the Game constructor function. It adds a property called `words` to the Game object being constructed and assigns an empty object to it on line 2. It then iterates through a randomly chosen "game" (i.e. a randomly chosen set of letters and corresponding words to be found) and creates a Word object to represent each word to be found. Word objects draw themselves to the screen either as text or a placeholder when the word is yet to be found. The Word objects

are assigned to the words object by dynamically creating properties mapping the text of each word to its corresponding Word object.

```
1  var game = games[config.difficulty][i];
2  this.words = {};
3  for (var i = 0; i < game.words.length; ++i) {
4    var word = game.words[i];
5    this.words[word] = new Word(word);
6  }
```

Listing 9: *A user-defined object demonstrating dynamic properties*

JavaScript also provides special support for using objects as arrays. When an object is created using the array literal syntax [] or Array constructor, `Array()` and property names are integer values, the language provides a `length` property on the object that is automatically updated to reflect the number of elements in the array.

Lua: Tables Lua offers a single, fundamental data structure: *table*. Tables are the only data structure in Lua. All the other structures that the language offers can be represented as Tables efficiently. In traditional languages like C and C++, most of the structures are represented with arrays and lists. However, in Lua, tables are more powerful than either. The algorithms used for implementing these structures in traditional languages are simplified with the use of tables. Tables offer direct access to any type. Tables are fundamentally associative arrays, i.e., they are key-value pairs in which the keys can be any legitimate value in Lua. Tables are quite similar to the object type of JavaScript, the difference being that with JavaScript objects the keys can be only either strings or integers, whereas in Lua tables the keys can be any value in Lua except nil. We now describe the different structures in Lua that we defined using tables. Tables are very easy to create. An empty table can be assigned to any variable by specifying empty curly braces which denotes an empty table constructor. For instance, throughout our code, we have declared tables easily and efficiently as shown in the following code.

```
1    games_letters = {}
2    letters_guessed = {}
3    solutions = {}
4    words_guessed_correct = {}
```

Listing 10: *Declaring tables in LuaLetterlizard*

Arrays in Lua can be implemented efficiently using Lua tables by simply indexing the tables with integers. Hence, arrays do not have a finite size, instead they can grow in size as needed. The following code snippet is an example of

an array implementation from Lua Letter Lizard. By convention, array indexing starts from 1 in Lua.

```
1     games_words = games.easy[i].words
```

Listing 11: *An array, `games.easy`, that holds pre-generated game puzzles and is easily accessed by indexing into the table*

Metatables are a very powerful feature offered by Lua. Metatables allow us to change the behaviour of a table. For instance, using metatables we can define how Lua computes the expression `a+b` where `a` and `b` are tables and establish prototype chains like we have in JavaScript.

2.2 Variable Scope

Overall, we discovered that Lua and JavaScript have many more similarities with each other than they do with Python; however, variable scope is one area where this differs. Both Python and JavaScript implement function scoping. The only means to create a scope for a variable in Python is a function, class or module and only a function in JavaScript. The official Python documentation states: "If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block [8]." This subtle point can lead to surprising results, especially for programmers not intimately familiar with the scoping rules in these languages. Consider the following example:

```
1 a = 42
2 def f():
3   print(a)
4   if (True):
5     a = 43
6 f()
```

Listing 12: *A demonstration of function scope in Python*

Most programmers familiar with the block scoping rules of C, C++, and Java would expect this code to print 42; however, it results in an `UnboundLocalError` because the assignment on line 5 creates a new local variable called "a" that shadows the variable with the same name at the global scope and is visible throughout the entire function, but when line 3 is executed, this variable has not yet been assigned a value. Also, because Python lacks an explicit variable declaration statement, it is not clear whether the assignment on line 5 was intended to create a new local variable or change the value of the global variable. In fact, prior to Python 3, there was no way for code in an inner scope to assign a value to an enclosing scope that was not the global scope. Some developers call

this "broken lexical scoping" and Python 3 fixes it with the introduction of the `nonlocal` statement.

JavaScript uses function scoping for variables, similar to Python, which means that all variables declared in a function are visible throughout the entire body of the function. Unlike Python, though, JavaScript has an explicit variable declaration statement, which is known as "hoisting:" JavaScript code behaves as if all variable declarations in a function are hoisted to the top of the function. In both languages, this feature can easily cause bugs that are hard to find when local variables shadow variables in an outer scope, and for this reason it is considered a negative feature [2].

On the other hand, Lua implements block scope similar to C, C++ and Java. The body of a control structure, body of a function, or chunk (a segment of code that is treated as a unit) all introduce new scopes. Additionally, new scopes can be created explicitly with the keywords `do` ... `end`. The scope of a declared variable begins after the declaration and goes until the end of the block. When written in Lua, as shown below, the previous example will print 42.

```
a = 42
function f()
  print(a)
    if (true) then
        local a = 43
    end
end
f()
```

Listing 13: *A demonstration of block scope in Lua*

2.3 Functions

All three languages support first-class functions: functions can be stored in variables, passed as arguments to other functions, and be returned from functions as results. First class functions give these languages great flexibility: functions can be redefined to change their behaviour, or even erased to create a secure environment for untrusted code. Perhaps most importantly, functions can be nested and have access to the scope where they were defined. This means that functions in these languages are *closures* and this enables some important and powerful techniques that will be discussed next.

2.3.1 Closures

In addition to first-class functions, all three languages also support closures. Every function contains a reference to the scope chain that was in effect when the

function was defined, which is used to resolve variable names to values when the function is executed. A function, together with a reference to its scope chain is known as a *closure*. All three implementations make use of first-class functions and closures for implementing callbacks, especially the JavaScript version due to the event-driven nature of the client-side scripting environment in the Web browser.

The following example from LuaLetterLizard demonstrates the use of a closure for registering a set of callbacks to be called by the LÖVE game library when certain events occur. The variable `registry` is defined in `registerEvents` but also accessed in an anonymous inner function, which is made possible by the closure.

```
function GS.registerEvents(callbacks)
  local registry = {}
  callbacks = callbacks or all_callbacks
  for _, f in ipairs(callbacks) do
    registry[f] = love[f] or __NULL__
    love[f] = function(...)
      registry[f](...)
      return GS[f](...)
    end
  end
end
```

Listing 14: *A closure in Lua*

The next example is from LetterLizardJS. A callback function is assigned to the timer object's `ontimeup` property that is called when the time remaining reaches zero. This code snippet is contained within the Game constructor and the inner function needs access to the Game objects, so we first store the value of this (which refers to the Game object) in a variable called `that`, which will be contained within the closure and accessible to the inner function.

```
  var that = this;
  this.timer.ontimeup = function() {
    timeup = true;
    for (var word in that.words) {
      that.words[word].show(true);
    }
    that.nextRoundOrEnd();
  };
```

Listing 15: *A closure in JavaScript*

2.4 Object-Oriented Programming

All three languages take a different approach to object-oriented programming. Python supports a traditional class-base inheritance model, while JavaScript supports a protoype-based inheritance model.

Python Python utilizes object-oriented programming in a fairly typical sense. It utilizes the `class` keyword in order to declare a class. It combines OOP constructs from C++ and Modula-3. In Python, one can utilize multiple base classes. Derived classes are able to override any methods of its base classes, and methods can call methods of base classes with the same name. In Python, to set a variable `x` to be equal to a new instance of a class `Person`, one can simply write `x = Person()`. Python does not use the `new` operator. [7]

Python utilizes the convention that the variable `self` as the first argument of instance methods of a class, will essentially act as the `this` keyword often found in other languages. The `self` keyword can be replaced with any word of one's choosing. Some may find this construct a bit clumsy, as one has to repeatedly type the same variable over and over again in the beginning of the argument list of each method.

Unlike languages such as Java or C++, Python does not have any true concept of private variables. Rather, Python encourages a culture where programmers are "trusted" to not access variables they are not privy to. A convention that has been established is to use underscores at the beginning of a variable name to denote that it is a private variable.

JavaScript JavaScript supports object-oriented programming, but not in the classical sense. Rather than providing class-based inheritance, JavaScript provides *prototype-based inheritance*. In addition to having their own set of properties (known as "own properties") every object is associated with and inherits properties from another object called its *prototype*, indicated by its prototype property (occasionally the value of the prototype property will be null, but this is rare).

Many programmers who are new to JavaScript who come from a C, C++, or Java background find JavaScript's prototype-based inheritance confusing at first; however, it is really quite simple and works nicely with JavaScript's dynamic nature. If a requested property is not an own property of an object, then its prototype, its prototype's prototype, etc. are searched recursively until the property is found. Most classical object-oriented features including static class-based objects, structs and inheritance hierarchies can be easily simulated in JavaScript.

Prototypal inheritance is a key feature of JavaScript, and LetterLizardJS makes extensive use of objects and protoypes. The following shows the definition of

the Tile constructor function and its prototype object. The constructor function is used to create a *class* of objects that have their own properties to hold their state, but share behaviour inherited from the prototype object:

```
function Tile(letter) {
  this.letter = letter;
  this.saved = {};
}

Tile.prototype = {
  placeAt: function(x, y) {
    this.container.x = x;
    this.container.y = y;
    this.saved.x = x;
    this.saved.y = y;
  },

  moveTo: function(x, y, save) {
    // Moves the tile to the specified x and y coordinates.
    // This function is called by the Scramble and Builder classes
    createjs.Tween.get(this.container).to({x:x, y:y}, 500);
    if (save) {
      this.saved.x = x;
      this.saved.y = y;
    }
  },

  reset: function() {
    this.moveTo(this.saved.x, this.saved.y);
  },
};
```

Listing 16: *A class definition in JavaScript*

Lua Lua does not directly implement object-oriented features, but these can be simulated using tables. Similar to objects, tables have a state and identity independent of their values. Two tables with the same value are two different objects. We can store different values within a table field and access it using the dot operator. Lua tables also have support for the `self` parameter which tells the method on which object it has to operate. With the use of the self parameter, the same method can be used to act on many objects. Lua adds a hidden self parameter with the colon operator. There is no notion of a 'class', but prototype based inheritance can be implemented using metatables which are similar to JavaScript objects but are more powerful.

3 Conclusion

We have gained a deeper understanding and appreciation for scripting languages by implementing a non-trivial program, the Letter Lizard game, in three different scripting languages and comparing the implementations. We found that it was fairly easy to implement the game in all three languages, and that the effort required was less that would have been required to implement the game in a static language such as C, C++ or Java. Of the three implementations, we found that JavaScript required the least amount of effort due to the high-level functionality and event-driven nature of the client-side scripting environment provided by Web browsers. Of all the features that we used, we found first-class functions and closures to be the most useful. First-class functions and closures made writing callback functions for event-driven programming simple and easy, and led to a clean, modular design. We avoided some of the negative features of JavaScript, such as automatic semicolon insertion, by always finishing statements with an explicit semicolon. Although JavaScript objects are not as powerful as Lua tables, we did not encounter any limitations in our implementation. Similarly, although Python's "broken lexical scoping" may cause problems for some programs, we did not encounter any difficulty with our implementation. Overall, we gained a deeper understanding of, and appreciation for, scripting languages and their ability to enable rapid application development and we will likely use scripting languages more often in the future.

References

[1] About Python. `https://www.python.org/about/`, 2014. Python Software Foundation.

[2] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., Sebastopol, CA, May 2008.

[3] Getting Started - LOVE. `http://http://www.love2d.org/wiki/Getting_Started`, 2014.

[4] Getting Started with Python. `http://python.about.com/od/gettingstarted/ss/whatispython_3.htm1`, 2014. About.com.

[5] node.js. `http://nodejs.org/`. Joyent, Inc.

[6] The Programming Language Lua. `http://www.lua.org`. PUC Rio.

[7] Python Classes. https://docs.python.org/2/tutorial/classes.html, 2014.

[8] The python language reference: 4. execution model. https://docs.python.org/2/reference/executionmodel.html, 2014. Python Software Foundation.