AN OVERVIEW OF LUA

Jade Marmorstein
jhmarmorstein@email.arizona.edu

Thomas Ruff
truff@email.arizona.edu

April 21, 2020

ABSTRACT

1 Introduction

2 History

2.1 The origin of Lua

Lua was created in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The three worked together as faculty, post-doc and PhD candidate respectfully at Tecgraf, the Computer Graphics Technology Group of the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in Brazil.

In their paper "The Evolution of Lua" [1], the creators provide context to the climate that led to Lua's birth. At that time period in Brazil, the country "had a policy of strong trade barriers...motivated by a nationalistic feeling that Brazil could and should produce its own hardware and software." (2) Due to this, Tecgraf did not have the means to utilize hardware or software from abroad. Not only financially, but because in that political climate they would have had to prove through a laborious bureaucratic process an extraordinary reason why a Brazilian company couldn't fulfill their needs. As a result, Tecgraf was forced to develop most of its tools in-house, from scratch.

One of Tecgraf's large clients was an oil company. The company required a language for front-end data entry, and for creating graphical programs, and so the predecessors to Lua, DEL and SOL, were created. While DEL and SOL were successful in serving their initial purposes, the users gradually required more complex tasks and more power and sophistication from the languages. Ierusalimschy, Henrique de Figueiredo and Celes convened to meet this growing need. At the time, no good scripting languages for their needs existed - most were disregarded for unfamiliar syntax or being too underdeveloped. As such, the only option seemed to be to create their own, new language. And thus, Lua was born.

2.2 Current Status of Lua

Lua has become a popular scripting language for a variety of different uses. However interestingly, it has perhaps become most widely used for game development. In fact, Lua was reported in the early 2000's to have become the most popular scripting language for game development.This was not the original intent for the language, but the creators have reflected that it makes sense and cite features of Lua such as its portability, simplicity, and ease of embedding that make it so useful for game development. They also provide examples of games developed in Lua, including Baldur's Gate, Escape from Monkey Island, The Sims, and World of Warcraft to name a few. Fans of games such as Elder Scrolls Online have also widely used Lua for scripting mods to customize the gameplay experience. On the more dubious side of things, it is also the language used to create Cheat Engine, an application for cheating in computer games. Beyond games, Lua scripting has been used for a variety of purposes such as by Adobe to create the UI for Adobe Lightroom, and in TI calculators. All in all, Lua stands currently as a popular language with an active community.

3 Control Structures

Lua's control structures follow many conventional standards in terms of functionality. While the common conditional and iteration structures work just as found in other languages, Lua's syntactical systems favor an English implementation using the words "do", "then", "end", etc. as opposed to a more traditional symbolic syntax using curly brackets. A standard Lua script has access to the conditional statement, both for and while loops, recursion, goto statements, a repeat until structure, and several others.

Lua does offer an alternative to the traditional while loop in the form of the repeat until structure, which performs the code within the repeat block and checks the until condition. If the until condition is false, the repeat block is executed again. Otherwise, execution proceeds beyond the until condition.

```
-- print the first non-empty line
   repeat
     line = io.read()
   until line ~= ""
   print(line)
```

One notable omission from the standard selection of control structures in Lua is the popular switch statement found in C and other languages. The behavior of a switch case expression can be achieved using a conditional chain however, so the functionality is still present in the language. The iterator function table data structures allow for clean expression of switch expressions without the need for a dedicated control structure.

Lua's for loops utilize a similar functionality to Python and other languages, having an integration between the standard start, stop, and step for loop and the iterator for loop. This flexibility in implementation but simplicity of expression allows users to quickly and easily develop Lua scripts, without needing to consult documentation as frequently to build control systems and program flow.

While typically frowned upon by many developers, Lua does support the goto statement with labels to allow unconditional jumps in the script. Denoting a label with the ::label:: syntax, a goto label statement will immediately move execution to the location of the given label. Despite being unpopular with many programmers, Lua does allow these statements for compatibility and similarity with its parent, C.

4 Data Types

As the language reference manual specifies, there are eight data types in Lua. [2] They are: nil, boolean, number, string, function, userdata, thread, and table.

The first five of these data structures are relatively straightforward, as they are the same or variants of data types that are common in most languages. nil is simply the same as null, though the reference manual notes that is the only value other than the boolean value "false" that will cause a condition to evaluate to false. In general though, it indicates the absence of a value. The boolean type, of the values "true" or "false", is the same is it occurs in most other languages. Number is also rather self-explanatory, though unlike many other languages which have separate types for int, float, double etc, number is simply an umbrella type that encompasses all numbers. A number in Lua can be an integer, a decimal number, and can even be written in scientific notation such as 3.2e14. Strings, also just as in any other language, are a sequence of characters enclosed in quotation marks such as "string". The usual escape characters in Lua as most languages apply as well, such as '\t' for tab and '\n' for new line. Functions also work as they do in other languages, but can be considered a data type as a variable can be assigned a function value, and passed as a parameter to another function.

The next three data types in Lua are a little less common. It is worth noting these three types are objects, and contain references rather than values.

The userdata data type corresponds to a block of memory. It has no pre-defined operations in lua and exists to allow for C data to be stored in Lua variables. This is not a commonly used type and is mostly for programmers who want to integrate C programs in Lua.

The thread type is more like its counterpart in other languages. Threads represent independent processes that can run concurrently in multi-threaded programs. Threads are used to implement coroutines, essentially a way of having multiple independently executing processes.

The final and most important data type in Lua is tables. Tables are used to implement data structures of all kinds. As the reference manual for the language states, "Tables in Lua are not a data structure; they are *the* data structure" [2]. Tables are associative arrays which can be indexed with any value, not just numbers (with the exception of nil), and can contain any type. They are the sole data structure in Lua and can be used to implement arrays, lists, linked lists, queues, stacks, trees, and pretty much any other data structure one can think of from any other language. Tables can be declared like typical arrays, for example:

```
array = {a, b, c}
```

and the values can be indexed into normally like array[i]. However, it's important to note that unlike most languages, Lua tables start with a base index of 1 rather than 0. Tables can also be implemented like the following:

```
list = {next = list, value = 30}
```

This implementation is more similar to a python dictionary, but can be used to implement other data structures. The above example would be a way to implement a linked list in lua. Each element of the table can also be referenced with a dot operator like list.value, which is essentially syntactic sugar for list[value].

5 Subprograms

The primary system of breaking up larger programs into smaller, subdivided sections of code in Lua is the built in module system, which allows users to compile Lua files into large tables that contain a number of functions and variables that perform some common function. In a way, Lua modules behave similarly to libraries in other languages like Python or Java.

A module in Lua can be loaded into another Lua file for use using the "require" keyword, like so:

```
mymathmodule = require("mymath")
```

This would load the Lua module named "mymath" and be accessible under the name "mymathmodule", a structure very similar to Python's import system. Using this system of importable modules, larger scale Lua projects can be broken down into different files containing functions needed for the full functionality of the program. This abstraction allows for multiple different developers to produce modular code usable outside of the project it was originally written for.

Object oriented programming is possible in Lua thanks again to the table data structure. Similar to how a table containing functions and data can be used to produce an independent module usable by other Lua files, a table can also hold state and behavior of an object. But in order to fully produce object oriented behavior, Lua must be able to define how multiple objects interact with each other, which is achieved using the system of metatables.

Metatables in Lua are just as the name suggest: tables written about tables. A metatable is a table used to describe how a table interacts with another table. Metatables contain a number of "special keys" which are used to hold functions that describe an override to the standard result when table operations are performed. For example, if the metatable contains the (key,value) pair ("__tostring", myToString), the function myToString will be called instead of the standard tostring method used for tables. These structures can be used to produce object behavior by providing unique implementations to standard operations like the above tostring example, mathematical operations, or other methods.

While such object oriented behavior is available in Lua, standard practice prefers imperative programming with minimal object usage outside of game development. There is no concept of classes in Lua, and while inheritance behavior can be achieved through the use of metatables and first-order functions, Lua programmers tend to prefer the procedural paradigm of Lua's parent language C. By design, Lua was intended to be embedded in other applications, so the relatively straightforward nature of procedural programming is most applicable.

6 Summary

References

References

[1] Ierusalimschy, Roberto. Henrique de Figueiredo, Luis. Celes, Waldemar. *The Evolutin of Lua* lua, https://www.lua.org/doc/hopl.pdf

[2] Ierusalimischy, Roberto. Henrique de Figueiredo, Luis. Celes, Waldemar. *Lua Reference Manual* lua, https://www.lua.org/docs.html

[3] *r/lua Reddit* Reddit, https://www.reddit.com/r/lua/

[4] *Cheat Engine* Cheat Engine, https://www.cheatengine.org

[5] *Cheat Engine Repository* GitHub, https://github.com/cheat-engine/cheat-engine/

[6] *ZeroBrane Studio* ZeroBrane, https://studio.zerobrane.com