universität innsbruck

Leopold-Franzens-
Universität
Innsbruck

Lua: A programming language analysis

Author: Lukas Niederstätter
Matrikel-Nr. 11916084

VU Specialization Seminar

Associate Professor: Prof. Cezary Kaliszyk

Summer Semester 2023

17.06.2023

Contents

1 Abstract

Developed in the 1990s, Lua is a powerful and lightweight scripting language and has become an indispensable tool for software development in recent years. It stands out for its minimalist syntax and fast execution speed, making it ideal for integration into other software projects. Embedded into existing systems, Lua provides high performance and scalability, allowing it to find application in wider areas such as embedded systems, web development, video games, data analysis and more. In this paper, we report at the history and evolution of Lua, the necessity of its creation, strengths and limitations compared to more common-known programming languages.

2 Introduction

Lua, the scripting language, emerged in 1993 within Tecgraf, a research and development laboratory at PUC-Rio in Brazil. Its creators, Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, had diverse backgrounds. Roberto was a computer scientist specializing in programming languages, Luiz Henrique was a mathematician focused on software tools and computer graphics, and Waldemar was an engineer interested in the applications of computer graphics.

Tecgraf, known for its collaboration with industrial partners, initially focused on developing basic software tools to meet the graphical program requirements of its clients. Due to Brazil's trade barriers during that time, Tecgraf's clients faced difficulties obtaining customized software from abroad. As a result, Tecgraf had to develop its own tools to fulfill their needs. One of Tecgraf's significant partners, Petrobras, the Brazilian oil company, relied on Tecgraf for interactive graphical programs in engineering applications. They had already created two little languages, DEL and SOL, to serve specific purposes within Petrobras. These languages laid the foundation for the development of Lua.

3 History and Creation

The engineers at Petrobras were tasked to prepare files filled with numbers, called input files. These would then be used to execute numerical simulations multiple times a day. The input files needed to be strictly formatted which made their task error-prone and repetitive. In early 1992, Petrobras approached Tecgraf to develop a graphical solution to simplify this process.

Tecgraf's solution involved the creation of a dozen graphical front-ends that allowed the engineers to input the numbers by clicking on the relevant parts of the simulation diagram. To streamline the development, Luiz Henrique de Figueiredo and Luiz Cristovao Gomes Coelho led a team that adopted a uniform coding approach, resulting to the development of DEL.

3.1 DEL

DEL is a simple declarative language that described each data entry task. In a DEL program, entries were defined with each entry having named and typed fields.

```
:e      gasket              "gasket properties"
mat     s                    # material
m       f       0            # factor m
y       f       0            # settlement stress
t       i       1            # facing type

:p
gasket.m>30
gasket.m<3000
gasket.y>335.8
gasket.y<2576.8
```

Figure 1: Code Snippet of a small DEL program [1]

In this code section, entries are created and saved as entities defined under the statement ":e". Every entity has a name and fields initialized with values. Below we have the statement ":p" which defines a section of predicate logic. This allows for basic data validation as well as basic input rules.

While an entity in DEL could be associated as record or structure in some other programming languages, there is one essential feature which made DEL suitable for Tecgraf's problem: Every entity produced a graphical meta-file with its name and the diagram in which the engineers entered the data. This allowed the graphical programs to read those files which could then be visualized for Petrobras with every aspect needed.
Soon, users would require even more from DEL, such as boolean expression, conditional controls or even loops which the language was not suited for. This led the developer in Tecgraf to come up and develop a greater programming language.

3.2 SOL

SOL stands for simple object language and heavily draws design ideas from BibTex[2] and UIL. It was created for a different project at the same time DEL was developed. This project consisted of generating reports, having different columns as well as different fonts, size and options. Users would be able to create and modify these tracks and each report would then be saved as file for reuse purposes. Instead of designing a language that incorporates the objects needed, SOL was designed with type declaration in order to handle every different type of reports.

```
type @track{ x:number, y:number=23, id=0 }
type @line{ t:@track=@track{x=8}, z:number* }
T = @track{ y=9, x=10, id="1992-34" }
L = @line{ t=@track{x=T.y, y=T.x}, z=[2,3,4] }
```

Figure 2: Type Declaration in SOL [1]

We can observe some similarities between the creation of entities in 1 and 2. Both entities have fields and values initialized as soon as the entity gets declared however SOL is able to declare the type of the entity, which can be a track, a line or another option and having their fields and values integrated in said declaration.

4 Concepts and Versions

Although SOL's development was successful, it never found usage as the developers decided not to implement it. Instead, they combined both languages to create a more versatile and more portable language. Lua 1.0 was born. In *The Evolution of Lua*, the authors reasoned:

"Given the requirements of ED and PGM, we decided that we needed a real programming language, with assignments, control structures, subroutines, etc. The language should also offer data-description facilities, such as those offered by SOL. Moreover,because many potential users of the language were not professional programmers, the language should avoid cryptic syntax and semantics. The implementation of the new language should be highly portable, because Tecgraf's clients had a very diverse collection of computer platforms. Finally, since we expected that other Tecgraf products would also need to embed a scripting language, the new language should follow the example of SOL and be provided as a library with a C API"[3].

4.1 Lua 1

Combining both DEL and SOL was very successful and soon started to gain attention in both Tecgraf and Petrobras as they identified Lua's potential to support graphic metafiles. Since it was a procedural language with type declaration, those files were not relying on other languages to model those object. Lua 1.0 was born. As its user base started to grow, many more demands were requested, the first one being for better performance, even with bigger and heavier programs. One of Lua's first update was its compiler. Since all programs were precompiled to bytecode, this meant that it had to run and compile fast. By removing the lex-generated scanner by a self-written one, the developer achieved almost twice the original compiling speed. They also create new compiler codes to decrease the amount of compiler instruction and created a new instruction to help pushing elements from the stack. Instructions of the new compiler looked like this:

```
CREATETABLE
PUSHNUMBER 30              # value
PUSHNUMBER 40              # value
PUSHNUMBER 50              # value
SETTABLE 1 3               # set elements from index 1 to 3
```

Figure 3: Compiler Instructions [1]

The new instruction "SETTABLE" allowed all elements within the given index to be added to the list with only one instruction instead of using two separate instructions for every single element. This procedure was able to cut instructions in half and also decreased compile time. These changes were released as version 1.1 but under a restrictive license. This would allow using Lua under academic purposes but would prevent commercial usage. This would prove to be as a big downside for the language as, during this time, languages like Perl were free to use.

4.2 Lua 2

This version was released in early 1995 and was published as free software, lifting all restrictions from the previous versions. This version, 2.1, brought many important changes for the language as the developer were still focused to increase the simplicity and decrease compiling time. This caused the first major incompatibilities to older versions. Removing the "@" from the constructor assignments was one particular important change as it simplified a lot of the syntax and the semantics of defining tracks. Another important update was the introduction of fallback functions. Users had now the ability to write such functions which are called whenever Lua is stuck or didn't know how to proceed in the program. Lua has now become an **extensible** language. The first fallbacks were designed for basic arithmetic and comparison but soon would include more functionalities such as string concatenation and table access. Especially for table access, this was revolutionary as table access in previous version was needed to be exact. With the now

added fallbacks, users could define functions for every eventuality or mistakes happening during the access.

The ability for users to define these fallback functions themselves as compared to hard-coded fallback functions would not only allow the users to write functions in a way they need them and only for functionalities they would actually use in that moment, this would also allow Lua to support object-oriented programming with the ability to write method for inheritance and operator overloading. This so called "meta-mechanism" changed the main design pattern for Lua and in May 1996, these features would be released as Lua 2.4. Additionally, this version featured a new compiler called **luac**.
Luac is a external compiler which saves the precompiled code and string tables to a binary file. This binary file was portable and could avoid parsing and code generation at run-time. This allowed for low compiling times even for bigger and more costly programs. Besides portability, luac introduced a syntax check and source code protection. However, this compiler did not imply fast execution speed. This was merely a save mechanism for a file to be executable later. In November 1996, Lua 2.5 was released with its own pattern-matching engine. This brought two new functions forward. "strfind" was able to find any string given to the function and "gsub" was able to replace substrings matching a given pattern in a large string. Although there were only a few updates, this version proved to be the one with the most success on an international level. The very same year an academic paper was published and Lua made its appearance in many different magazines and research papers. Lua was even more praised in 1997 as it made its first use as programming language for game development. In *The evolution of an extension language: a history of Lua*, the authors wrote:

"This first use of Lua in a game attracted the attention of many game developers around the world to the language. Soon after, Lua started to appear frequently in game newsgroups, such as rec.games.programmer and comp.ai.games" [1].

4.3 Lua 3

Within the same year, Lua 3.0 was released, now replacing its fallback methods with tag methods. Users could now create tags and associate tables and user data to them, essentially allowing user to create different fallback methods for different tags. At first these tags were seen as an exception-handling mechanism but soon they were actually used to represent user-created types. This further boosted object-oriented programming since now every tag could have its own fallback methods rather than having them globally for every tag created. Another important feature of Lua 3.1 was the introduction of functional programming. Users were now able to create anonymous functions and function closures. Previous, higher functions and functions which take functions as parameter, like "gsub", have benefit by this implementation. Newer version like 3.2 didn't bring much change to the language as they were only maintenance releases. The C API, on which Lua is based on, remained in tact and would not experience changes until Lua 4.0.

4.4 Lua 4

This version brought major changes as Lua would receive a new API released in November 2000, introducing multiple Lua states for applications as well as clear stack exchanges between Lua and C. Lua 4.0 would also introduce for-loops for the first time. In the paper *The Evolution of Lua [3]*, the authors explained that they couldn't agree on its syntax as well as users forgetting to update their control variable in while-loops which led to infinite loops. The developers finally settled on implementing two different for-loops: A single, numeric for-loop as well as a table-traversing for-loop, comparable to foreach-loop in other language.

4.5 Lua 5

Lua 4.1 was next in line but due to the many changes, the developer decided to have it be version 5.0 instead. Implementations like threads, full lexical scoping and the substitution of fallbacks with newly introduced, generic for-loops replacing all fallback methods and tags were now available to use. Over the years, in version 5.1, Lua would also receive a module system as well as an upgrade to its garbage collector. With the latest release being Lua 5.6.4, not much has changed from its previous versions as syntax and semantics remained the same.

5 Syntax and Semantics

Lua's syntax was designed to be simple, short and easy to understand/write so users can learn the language faster. One method of its fast execution is applying all line statements into a chunk. A chunk can be composed from a single line instruction to any number of instructions bundled together. This is not uncommon as the interpreter has no problem with larger chunks. Some syntactic and semantic features are listed in this section, but not as many of them as most of it draws close parallels to Modula and C.

5.1 Basic instructions and statements

```
a = 1
b = a*2

a = 1;
b = a*2;

a = 1 ; b = a*2

a = 1   b = a*2
```

Figure 4: Basic statements [4]

The following instructions are all valid as Lua neither requires semicolons or line separation to have them executed. Basic functionalities would also rely on being collected to a chunk in order for the interpreter to execute them faster. Another common was was just to use the stand-alone interpreter in the interactive mode. In this mode, Lua would mostly have single-line chunks but not necessary as the interpreter could detect if a chunk was not completed and therefor waits for more input to come. In this figure, all declarations are local. To create a global variable, one has just to define it since there is no error when accessing a variable or field that has not been initialized yet.

5.2 Running Files

Lua has a specific way of calling files. Resembling the language C, the stand-alone interpreter requires file inclusion with the method "dofile()". This method loads the given ".LUA" file and now allows the interactive mode to access all of its functions and

fields. This proves to be very effective when it comes to testing as you can include the files via this method and then print the results of functions, fields, etc. without debugging or writing tests for it.

5.3 Comments and Lexical Conventions

Lua has a handful selection of keywords, comparable to C or Java, like: **and, end, break, true, false, while, for, etc.**. These can not be used as identifier, however Lua is case-sensitive which means identifiers like "AND" or "And" is valid.

```
--[[
print(10)         -- no action (comment)
--]]
```

Figure 5: Comment and Comment Blocks [4]

In this figure we can observe two types of comments. Lua is also handling false commenting very well. If you would add another "-" to this figure's comment block, the instruction remains valid as the compiler simply removes the falsely set comment block, execute the instruction and would handle the line below as single line comment.

5.4 Types, Values and Operations

Lua has a total of eight different types: *nil (Compared to NULL in other languages), boolean, number, string, userdata, functions, table and thread.* Like mentioned before any non-initialized variable or field has automatically a value of nil and Lua does not interpret it as an error. This is also helpful to differentiate return values of a function as nil-values can simply used for error-cases or wrong-conditioned execution.

Just like C, Lua also covers basic arithmetic as well as logical and relational operations. Another useful integration, this time applied from Python, Lua uses "self" calls when creating and assigning tables to a variable. Assume a table variable **tab1**. This now would mean you can access its fields via **tab1.self and tab1.x, tab1.y**.

5.5 Constructors

Lua offers constructors to create and initialize tables. The simplest constructor form is the empty constructor. A big advantage of constructors would be the limitless use of expressions within since it does not need to use only constants or constant expressions. Also typing of expressions and constants can be chosen at will.

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"}
```

Figure 6: Basic Constructor Instruction [4]

5.6 Functions and Multi-Return

Functions are similar to Python Syntax: Functions can have multiple return values. This tends to work by listing all return values after the return keyword to have them returned back. This proves to be very efficient and versatile when you need more information for the main returning values, be it for getting the index of a number in a list or finding more values which other languages may need more code instructions to do so.

```
function maximum (a)
  local mi = 1          -- maximum index
  local m = a[mi]       -- maximum value
  for i,val in ipairs(a) do
    if val > m then
      mi = i
      m = val
    end
  end
  return m, mi
end

print(maximum({8,10,23,12,5}))     --> 23   3
```

Figure 7: Function with multi-return values [4]

6 Use-Cases in Modern Technologies

With Lua being easy to embed in other languages and technologies, its use-cases range from embedded scripting to web development and game development.

6.1 Game Development with Lua

Since its first usage in the 1990s, Lua proved to be very efficient in the game development sector since it has a great learning curve and had a fast execution speed, even with heavier files. Nowadays, many game and even modifications are written in Lua, with some games even featuring in-game coding in Lua to access/activate certain game mechanics. One of the best example would be a modification in the popular game ”Minecraft”.

There currently exists a modification called "OpenComputers" [5] where players have the opportunity to directly code in Lua within a small operating system within the modification itself. Not only can the players therefor impact many aspects of the main game, they also have the ability to modify aspects to their own ideas. Building control structures for artificial power generation, resource management and more is possible throughout this modification.

Many other games are written in Lua and over the years, Lua found its success in this section for being easy to learn, executing fast and being lightweight which allows it to be integrated into other software rather easy.

6.2 Embedded Systems with Lua

As written above, Lua's lightweight design allows users to embed it rather easily into software but also hardware as Lua is commonly used in handheld devices such as mobile phones and laptops for things like net filters and telecommunication processes and also finds its use in ethernet switches and microprocessors. Lua's scripting base allows it to be also used in other languages such as C, C and Python with the Luau library. This allows users to create Lua objects in said languages and applying so called "Lua-Sections" in order to execute Lua code within the language itself. C is the more common language to be used for this special feature as Lua was build on the C API. This would enable Lua to be integrated into C much better. There are some other abbreviations from Lua's original version like CGLua or JITLua which all have integrated an API of a major language to increase their power in their fields of use. JITLua is heavily used in web development while CGLua is used in game development.

7 Conclusion

Lua is a versatile and efficient programming language that has gained widespread popularity and adoption across various domains. Its lightweight nature, simplicity, and extensibility make it a suitable choice for a wide range of applications, from embedded systems to game development and scripting.
Lua's design philosophy of providing a minimalistic core with powerful extension capabilities has resulted in a language that is both easy to learn for beginners and powerful enough to meet the demands of experienced programmers. Its small footprint and fast execution speed make it ideal for resource-constrained environments and performance-critical applications.

One of Lua's standout features is its seamless integration with other programming languages, allowing developers to leverage existing code and libraries. Its C API provides a bridge between Lua and C/C++, enabling efficient interoperability and the creation of high-performance extensions.
Lua's emphasis on simplicity and flexibility has also led to the creation of an active and supportive community. The availability of extensive documentation, online resources,

and a wide range of libraries and frameworks further enhances the language's appeal and ease of use.

While Lua may not be as widely recognized as some mainstream languages, its unique blend of simplicity, performance, and versatility has made it a favorite choice for many developers and organizations. As a scripting language, Lua excels in providing a powerful yet lightweight solution for embedding in larger applications, enabling users to extend functionality and customize behavior.
In conclusion, Lua's strengths lie in its simplicity, efficiency, and extensibility, making it a valuable tool for a variety of programming tasks. Whether you're a beginner looking for an easy-to-learn language or an experienced developer seeking a flexible scripting solution, Lua has proven itself as a reliable and capable programming language.

References

[1] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of an extension language: a history of lua. Website, 2013. Reprint of Proceedings, online available at `www.lua.org/history.html`; visited 08.06.2023.

[2] Oren Patashnik. Bibtex 101. 1984.

[3] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1, 2007.

[4] Roberto Ierusalimschy. Programming in lua, 2003.

[5] CurseForge Inc. Opencomputers - minecraft mods. Website. Overview of OpenComputers available at `https://www.curseforge.com/minecraft/mc-mods/opencomputers`; visited 10.07.2023.