The Lua Programming Language

By Myles Duah, Alexander Leo, Daniel Lopez,
James Vannicola, and Kevin Zheng

SUNY Polytechnic Insitute of Technology
100 Seymour Rd, Utica, NY 13502, United States

December 10, 2022

Overview of Lua

The Lua programming language is a high-level, lightweight, and embeddable scripting language intended for general-purpose use. It was first designed and implemented in 1993 by a team of computer scientists at the Pontifical Catholic University of Rio de Janeiro in Brazil.

Lua's small size and ease of use are two of its defining characteristics. The entire Lua interpreter, including its standard libraries, is less than 400KB in size, making it an ideal choice for use in embedded systems or other environments where space is at a premium. Furthermore, Lua has a simple and easy-to-learn syntax, making it accessible to programmers of all skill levels.

Another distinguishing feature of Lua is its adaptability. It is intended to be easily understood and embedded into other applications, allowing those applications to be extended with scripting capabilities. This makes it an excellent choice for game development, where it is often used to provide scripting support for game logic and AI. Additionally, Lua is often used as a scripting language for web applications, providing a way to add dynamic functionality to websites without requiring the user to have any specialized knowledge of programming.

In terms of its performance, Lua is generally considered to be on par with other scripting languages such as Python and JavaScript. It is not as fast as compiled languages like C or C++, but its speed is sufficient for many applications. Additionally, Lua has a built-in garbage collector that automatically manages memory, making it easier to write efficient and memory-efficient code.

One of the modern applications of Lua is in game development. Many popular games, including World of Warcraft and Angry Birds, use Lua as a scripting language to provide support for game logic and AI. Additionally, Lua is often used as a scripting language for web applications, allowing developers to add dynamic functionality to websites without requiring users to have any specialized knowledge of programming.

Looking forward, it is likely that the use of Lua will continue to grow in both game development and web applications. Its small size, ease of use, and flexibility make it an attractive choice for a wide range of applications. Additionally, as the demand for more powerful and flexible scripting languages continues to grow, Lua is well-positioned to remain a popular choice for many developers.

Ultimately, the Lua programming language is a small, lightweight, and easy-to-use scripting language with a wide range of applications. Its flexibility and ease of use make it an excellent choice for game development and web applications, and its continued growth in popularity suggests that it will remain a popular choice for many developers moving forward.

History

Lua is a scripting/programming language originally developed in 1993 by three engineers named Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The engineers at the time of Lua's inception were employed in the Tecgraf department at the Pontifical Catholic University, known as PUC-Rio in Brazil. The language was created with the intent of being extremely lightweight, very portable, and easy to embed and it is referred to as an extensible language, meaning that it can be used to extend an application's use through end-user customization. Since its inception, Lua has seen extensive use in a wide range of application fields due to its previously stated high portability and lightweight nature, and its popularity only increases as it receives more updates. Lua 1.0 was complete and operating by July 28th, 1993, but saw no official public release as it was initially intended for use on PGM, a configurable report generator project being developed by Tecgraf for Brazilian oil company Petrobras. The unreleased 1.0 version of Lua contained constructors and garbage collection, 4 libraries, 5 built-in functions, 30 API functions, and 64 VM instructions. however, the data type boolean did not exist within the language at this stage and was alternatively substituted with data type Nil. The next iteration of Lua arrived on July 8th, 1994 dubbed Lua 1.1, which was the first official version of Lua to become publicly accessible. This version is virtually similar to 1.0, having the same features with an additional 2 built-in functions and 1 extra VM instruction for a total of 7 built-in functions and 65 VM instructions.

Version 2 of Lua was released on February 7th, 1995 with Lua version 2.1. Version 2.1 saw the inclusion of two major features: support for Object-Oriented Programming, and extensible semantics. Around the time of its release, object-oriented programming had skyrocketed in popularity, and despite the boom of object-oriented programming and demand from Lua users to add features from object-oriented programming, it was denied initially because doing so would affix a programming paradigm to Lua. Instead, flexible mechanisms were included in order to allow the user to create the model that was necessary for their use case. As the language gained popularity, new users began to notice that Lua itself did not have much in terms of debugging facilities. This was mitigated in release version 2.2 with the inclusion of a debug API after users' requests for such a feature grew continuously. Version 2.2 introduced long strings, the extended syntax for function definitions, and improved stack tracebacks. With the increasingly widespread use of Lua at Tecgraf came larger files that the language would need to work with. The larger and more complex the file was, the longer it would take to recompile and run. Lua 2.4 Introduced an external compiler known as Luac. This compiler allowed for offline syntax checking and boasted fast loading times. Included in this version was an improved version of the debug API introduced in Lua 2.2. New functions were included that allowed for user access to local variables and hooks and called them when necessary. The final release of Lua 2 was Lua 2.5, which was released on November 19th, 1996.

The following summer would see the release of Lua 3.0 on July 1st, 1997. Lua 3 was where time intervals between major releases were increasing, as it was felt that the language was maturing as a product. Lua 3 had three major versions: Lua 3.0 as stated before, Lua 3.1 (July 11th, 1998), and Lua 3.2 (July 8th,1999). Version 3.0 saw the inclusion of a new library known as “auxlib”, whose primary use case was to allow the user to create Lua libraries, Conditional compilation support, and tags and tag methods. Much like how Object-Oriented Programming was added back in 2.1, Functional Programming was introduced into Lua 3.1 by the implementation of anonymous functions and function closures using “up-values”. Lua 3.2 was largely considered a maintenance release, as it included no new major features and instead more on code cleanup and bug fixes. This version did include a new debug library, as well as new table functions but that was it feature-wise. Version 4.0 of Lua would see release on April 11th, 2003, and would see only bug fixes for updates during its lifetime. This version's main feature was a brand-new API that was used to write a new standard library to replace the built-in functions that

existed in the previous versions. Another main feature released with Lua 4.0 was the inclusion of "for" loops, which was the most requested feature at the time of 4.0's release. After Lua 4.0 was released, 4.1 began development, but due to the number of features that were in 4.1, the name was reconsidered. As a result, the final version of Lua 4.0 was 4.0.1, released on July 4th, 2002.

Lua 5 was released on April 11th, 2003 with version 5.0. This version was originally Lua 4.1, but as stated before was changed because the number of features within 4.1 during development caused the developers to reconsider the name. Lua 5 has five major releases: Lua 5.0, Lua 5.1 (February 21st, 2006), Lua 5.2 (December 16th, 2011), Lua 5.3 (January 12th, 2015), and Lua 5.4 (June 29th, 2020). Lua 5.0 brought multi-threading to Lua by use of co-routines, full lexical scoping (which replaced the up-values that were used for function closures) meta-tables (which replaced the tags and tags method), boolean, weak tables, an API for loading chunks of Lua code. Full lexical scoping was originally intended to be within Lua from the very beginning but was shelved until Lua 5.0 because a reasonable implementation in 1.0 could not be figured out. Version 5.1 brought with it incremental garbage collection, a feature heavily requested by game developers, a module system, the operator's mod, and length, updated syntax for long strings and comments, and meta-tables for all types. Version 5.2 included meta-methods, a new Global lexical scheme, a library for bitwise operations, Light C functions, Ephemeron tables, and the goto statement. 5.3 brought 32/64-bit system support, Integers (by default 64-bit), Bitwise operators, and a utf-8 library. Finally, Lua 5.4 is the most recent release of the language, with the current version at the time of writing being Lua 5.4.4 (January 26th, 2022). Compared to the previous Lua 5 releases, 5.4 was light in terms of new features, including constant variables, to-be-closed variables, and a new generational mode for garbage collection.

Distinguishing features

There are many features of Lua that distinguish it from other languages, however, the main feature is the use of 'tables' in Lua. Tables are a type of data structure defined in the Lua language that provides simple and efficient implementations for modules, prototype and class-based objects, arrays, lists, sets, and more. Tables in Lua have no fixed size and allow users to index the array not only using integers but strings and any other data type the Lua supports.

Tables are not values or variables but they are objects which are similar to objects in Java. These tables are created using the built-in constructor expression written as follows: { }.

Data types

Data type definitions do not exist within Lua, meaning that variables inside Lua programs are not assigned types. Instead, the type assignment is done at runtime by the interpreter based on the values that are entering the program. This is because in Lua, the values are what actually have types and whatever value is stored within a variable at runtime gets the type of the value. There are eight basic types found within Lua, these types are Number, Nil, String, Function, thread, Userdata, Boolean, and Table. Type Number holds any real integer or floating point number. Type String works as expected, it holds a string of characters. Strings in Lua are immutable, meaning that once a string value is entered into a variable, it cannot be changed. To change any string value, the original string value has to be copied, edited, and stored into a new variable. Boolean types in Lua function as intended, they can be either True or False. Additionally, the Boolean type has a conditional statement where it considers False and Nil both as False, and everything else is considered True. Type Nil is treated as a non-value, it represents a missing useful value. Another thing with nil is that global variables in Lua by default have nil value, and if there is an existing global variable in a program that is not necessary anymore, assigning nil to that variable will delete it. Type Functions are reserved for functions that are stored within variables. Since all values within Lua are considered first-class values The Userdata type is used primarily when working with C. This type allows arbitrary C data to be stored within Lua variables. This type is typically used when Lua is embedded into other programs.

Threads in Lua operate much like in other languages with one exception, the Coroutine. As a matter of fact, they are very similar, they both have their own stack, local variables, and instruction pointer. The difference is that while a program runs several threads concurrently to finish the job, Coroutines actually work in tandem, and its execution is only granted when it requests it explicitly. The threads type is used to implement Coroutines into programs. Finally, the Table type is used to implement dynamically allocated associative arrays. These are arrays that can be indexed with numbers, strings, variables, and everything else that has a value, with an

exception for type Nil. These tables can also be heterogeneous, meaning that they can contain values of all types, again with an exception for Nil. Within the tables, Nil is automatically associated with any variable that is not part of the table, and on the other hand, any variable with Nil already associated with it is not part of the table. Tables also do not have a fixed size, they can be as large as required. Tables are used to represent a wide variety of data structures such as (but not limited to) arrays and queues, in the most efficient and simple method available. Tables are not values nor variables in Lua, instead, they are treated as objects.

Portability

Lua is purposely a very portable language due to the core design of Lua being based on simpleness and how easy it is to implement Lua into other applications. Lua can run on all types of Unix, Windows, mobile devices, and more. The portability of Lua is part of what causes this language to be as good as it is. It allows for easy integration into applications containing other programming languages.

Primitive Operators

Lua offers the expected array of mathematical and logical operators in its core implementation. These include addition, subtraction, multiplication, division (+, -, *, /), and the unary operator: negation(-). Additional mathematical operators are available through the inclusion of specific libraries, mostly written in C, with one such example being the exponentiation (^) operator. This operator is not included by default, but the syntax for it is, so when Lua encounters this symbol in a mathematical expression, it resorts to what's known as a fallback. A fallback is, just as it sounds, an alternative procedure to follow when the default operation is insufficient to handle the situation. Using the fallback entails Lua calling the function 'arith' - which is included in the standard library - and passing three arguments: the two operands as well as the operator in string format ('pow'). This function returns the result of the function 'pow' as defined in C when called with these operands. This allows Lua to keep its core implementation as small as possible while still providing the functionality expected from basic mathematical operations.

Similar fallback behavior is used to handle cases where operands are not compatible with the operator being applied to them. For instance, if an attempt is made to perform addition on two string values, say '3' + '7', then Lua encounters a problem because the core math operators all operate on real numbers only. Lua gets around this by using its fallback function "arith" and passes the string representation of the numbers as well as the string "add". The function returns the result of the implied operation. These fallback functions are called only when necessary which helps Lua keep its efficiency as high as possible.

In addition to the mathematical operators described above, Lua offers relational operators to use in decision-making logic and program flow control. These include; less than, greater than, less than or equal to, greater than or equal to, not equal to, and equal to (<, >, <=, >=, ~=, ==). All relational operators result in a value of true or false. Most of these operators only operate on values of the same type. That is, comparisons of different data types using >, <, <=, or >= will result in an error being thrown. Despite the fact that Lua is dynamically typed and that there are fallback methods for the mathematical operations when using other types, there is no standard fallback mechanism for doing a logical comparison with non-standard types in Lua at this time. The exceptions to this rule are the == and ~= operators. They can be used with operands of different types however comparisons of this kind will always result in false. Therefore it is up to the programmer to ensure that comparisons are made between operands of the same type (except in the cases of == and ~=).

When comparing functions and/or tables using relational operators, only variables that point to the same exact object will be considered equivalent. As an example, the following code demonstrates that two functions - even if they are conceptually identical are not considered to be equal to each other:

```
a = function()print('hello world');end
b = function()print("hello world");end
print(a == b)          - - results in false
print(a < b)           - - results in error
The same goes for tables:
c = {1, 2, 3}
d = {1, 2, 3}
```

print(c == d) ***- - results in false***
print(c < d) ***- - results in error***
However, strings do allow for direct comparison using relational operators on an alphabetical basis:
E = 'hello world'
F = 'hello world'
G = 'jello world'
print(e == f) ***- - results in true***
print(e == g) ***- - results in false***
print(e < g) ***- - results in true (no error)***

Boolean operators are also available in Lua, which includes 'and', 'or', and 'not'. These operate on their operands from the point of view that the operands are either true or false. To accomplish this, all possible arguments must be classified into one of these categories, with everything except false and nil being classified as true. This is a departure from the usual convention that the numeric value of 0 is equal to false. In Lua, with 0 being a number of some sort, it is classified as true.

Boolean operations follow a short-cut evaluation scheme where the second operand is evaluated if and only if a conclusion can not be reached by evaluating the first operand alone. In the case of 'and', both operands need to be true in order for the entire expression to be true, so Lua initially looks at the first operand and if it is false, evaluation stops, and the return is set to false (or nil if that is the value of the first operand). The second operand is never evaluated, which saves processing time and power. If the first operand is not false (any value besides false or nil) then Lua simply returns the second operand. If that value is some non-false value, then the whole expression effectively evaluates to true and the actual value of the second operand is available for further use. In the case of 'or', only one of the operands needs to be true for the entire expression to be true, so Lua evaluates the first operand, and if it is not false, then it returns its value and discontinues evaluation of the second operand. If on the other hand, the first operand evaluates to false, then the value of the second operator is returned, whether it be true or false, it defines the result of the expression.

The precedence of operators is an important factor to consider when writing and reviewing code. In Lua, a fairly familiar order of operations type precedence is followed which includes all the different types of operators. The list is shown below in decreasing order:

^

Not

* /

+ -

.. (concatenation)

> < >= <= ~= ==

and

or

All operators are left-associative except for ^ and .. which are right-associative.

These semantics allow for the optimization of code by taking advantage of the returned value as shortcuts in the program flow. As an example consider the following code:

Max = (a > b) and a or b

This code always returns the greater value between the two variables. In the case of a > b, the first part of this expression evaluates to true.

Max = ***(true) and a or b***

Now following the order of precedence of operators, the 'and' is evaluated next, and since "true and something" results in "something, the next step gives us:

Max = a or b

In the case of 'or', only the first operand need be true for the expression to return true (actually the first operand) and so the entire expression resolves to variable 'a' which we stated in this example as being the larger of the two values. A similar logic could be followed for the case where a < b.

Scoping and parameter passing

Scoping and parameter passing are important concepts in the programming language Lua. Scoping refers to the visibility and accessibility of variables, and parameter passing refers to the

way in which information is passed between functions. Understanding these concepts is essential for writing effective and efficient code in Lua.

In Lua, there are three levels of scope: global, local, and upvalue. Global variables are those that are defined outside of any function and are visible and accessible throughout the entire program. Local variables, on the other hand, are defined within a function and are only visible and accessible within that function. Upvalue variables are similar to local variables, but they are defined in an outer function and are visible and accessible within an inner function.

One important aspect of scoping in Lua is that local variables take precedence over global and upvalue variables. This means that if a local variable and a global or upvalue variable have the same name, the local variable will be used within the function. This can be useful for avoiding conflicts and ensuring that the correct variable is being accessed.

Another important aspect of scoping in Lua is that upvalue variables are not subject to the same rules as global and local variables. Upvalue variables are not automatically created when they are first used, and they do not automatically become nil when they go out of scope. Instead, upvalue variables must be explicitly created using the local keyword and must be explicitly set to nil when they are no longer needed.

In terms of parameter passing, Lua uses a system known as "call-by-value." This means that when a function is called, the actual values of the arguments are passed to the function, rather than the variables themselves. This has a number of important implications for how information is passed between functions in Lua.

First, call-by-value ensures that the values of arguments are not changed within the function. If a function modifies an argument, the change will only affect the local copy of the argument within the function and will not affect the original variable. This can help to prevent unintended side effects and ensure that functions are more predictable and reliable.

Second, call-by-value also means that arguments are passed to functions in a more efficient manner. Since the actual values of the arguments are passed, rather than the variables themselves, the function can access the values directly without having to look up the variables in the scope chain. This can help to improve the performance of the program.

Finally, call-by-value also means that arguments are passed to functions in a more flexible manner. Since the values of arguments are passed, rather than the variables themselves, the function can accept any value that is compatible with the argument's type. This can help to make functions more versatile and adaptable.

In summary, scoping and parameter passing are important concepts in the programming language Lua. Scoping determines the visibility and accessibility of variables, and parameter passing determines how information is passed between functions. Understanding these concepts is essential for writing effective and efficient code in Lua.

I/O functionality

Lua is a powerful and fast programming language that is widely used for scripting purposes in various applications, from web and game development to scientific computing and artificial intelligence. One of the key features of Lua is its extensive support for input and output (I/O) operations, which allows it to interact with external sources of data and information and to control various devices and systems.

At the core of the I/O functionality of Lua is the io library, which provides a set of functions and methods for performing various input and output operations, such as reading and writing data from and to files, interacting with the operating system, and communicating with other programs and devices. The io library uses a file handle-based approach, where each I/O operation is performed on a specific file handle that represents an open file or device.

One of the most commonly used functions in the io library is io.open(), which is used to open a file or device for reading or writing. This function takes a file name or path and a mode

string as arguments and returns a file handle that can be used in subsequent I/O operations. For example, the following code opens a file called "data.txt" for writing and assigns the resulting file handle to a local variable called "file":

```
local file = io.open("data.txt", "w")
```

Once a file handle has been obtained, various methods can be called on it to perform different I/O operations. For instance, the file:write() method can be used to write data to the file represented by the file handle, while the file:read() method can be used to read data from the file. The following code demonstrates how these methods can be used to write and read a string to and from a file:

```
-- Open the file for writing
local file = io.open("data.txt", "w")

-- Write a string to the file
file:write("Hello, world!\n")

-- Close the file
file:close()

-- Open the file for reading
file = io.open("data.txt", "r")

-- Read the entire contents of the file into a string
local data = file:read("*a")

-- Print the string
print(data)
```

In addition to the methods provided by the io library, Lua also has a set of global functions for performing common I/O operations. For example, the print() function can be used to write data to the standard output stream, while the read() function can be used to read data from the standard input stream. The following code demonstrates how these functions can be used to write and read a string to and from the standard input and output streams:

```
-- Write a string to the standard output stream
```

```
print("Enter a string:")

-- Read a string from the standard input stream
local data = read()

-- Print the string
print("You entered: " .. data)
```

In addition to the standard input and output streams, Lua also allows for the creation of custom streams that can be used for reading and writing data from and to various sources and destinations. For instance, the io.popen() function can be used to open a stream to a pipe, which can be used to read and write data to and from an external program. The following code demonstrates how this function can be used to run the "ls" command and read the resulting output:

```
-- Open a stream to a pipe
local pipe = io.popen("ls")

-- Read the entire contents of the pipe into
```

Control structures

Control structures are a fundamental part of any programming language, allowing developers to control the flow of execution of their code. In Lua, control structures include conditional statements, loops, and other control structures that allow developers to create logical and efficient programs.

One of the most commonly used control structures in Lua is the if-else statement, which allows developers to perform different actions based on the evaluation of a given condition. This statement allows developers to check for certain conditions and execute specific code blocks based on the results of the evaluation. For example, the following code checks for the value of a variable and prints a message based on its value:

```
if variable == 1 then
print("The variable is 1")
else
print("The variable is not 1")
```

```
end
```

In this code, the if statement checks if the value of the variable is equal to 1. If it is, the first code block is executed and a message is printed. If the condition is not met, the else block is executed and a different message is printed.

Another common control structure in Lua is the for loop, which allows developers to iterate over a given set of values and perform an action for each value. This is commonly used to iterate over arrays or collections of data. For example, the following code iterates over a list of numbers and prints each value to the console:

```
for i = 1, 10 do
print(i)
end
```

In this code, the for loop is initialized with a starting value of 1 and an ending value of 10. For each iteration, the loop variable i is incremented by 1, and the current value is printed to the console. This code will print the numbers 1 through 10 to the console.

In addition to the if-else and for loop control structures, Lua also includes a while loop, which allows developers to repeat a code block until a given condition is met. This is useful for situations where the number of iterations is not known in advance, and the loop must continue until a certain condition is met. For example, the following code continues to prompt the user for input until they enter the correct password:

```
local password = "secret"
local input

while input ~= password do
input = io.read()
if input ~= password then
print("Incorrect password, try again.")
end
end
```

In this code, the while loop continues to run until the value of the input variable is equal to the password. Each iteration of the loop prompts the user for input and checks if it matches the password. If it does not, a message is printed and the loop continues. Once the correct password is entered, the loop exits and the code continues.

Aside from the basic control structures, Lua also includes a number of additional control structures for more complex situations. For example, the repeat-until loop is similar to a while loop, but it checks the condition at the end of each iteration instead of at the beginning. This allows the code block to be executed at least once before the condition is checked.

Additionally, Lua includes a goto statement, which allows developers to jump to a specified label in their code. This can be useful for breaking out of complex loops or control structures, or for jumping to a specific point in the code. However, the use of goto statements is generally discouraged in modern programming due to the potential for creating confusing and difficult-to-maintain code.

Overall, the control structures in Lua provide a powerful and flexible set of tools for controlling the flow of execution in a program. With these structures, developers can create logical and efficient programs that can handle a wide range of situations and inputs.

Data structures

Data structures are an essential component of any programming language, and Lua is no exception. In this section, we will explore the various data structures available in Lua and how they can be used to effectively store and manipulate data.

One of the most basic data structures in Lua is the table. Tables are used to store a collection of values, and they can be thought of as similar to arrays in other programming languages. Tables are created using the {} syntax, and values can be accessed using either their index or a key. For example, a table can be created and populated as follows:

local my_table = {1, 2, 3}

```
print(my_table[1]) -- Outputs 1

my_table["key"] = 4

print(my_table["key"]) -- Outputs 4
```

Tables can also be nested to create more complex data structures. For example, we could create a table of tables, where each sub-table contains data about a particular person.

```
local people = {
{
name = "John",
age = 25,
address = "123 Main St."
},
{
name = "Jane",
age = 30,
address = "456 Park Ave."
}
}

print(people[1].name) -- Outputs "John"
```

In addition to tables, Lua also has several other data structures, including strings, numbers, booleans, and nil. These data types are used to store basic information, such as a person's name or an integer value. Strings are created using double or single quotes, numbers are created by simply typing a number, and booleans are created by using the keywords true or false. Nil is a special value that represents an absence of data.

Another important data structure in Lua is the function. Functions are blocks of code that can be called from elsewhere in the program to perform a specific task. They are created using the function keyword, and they can take one or more arguments and return a value. For example, we could create a function that takes a person's name and age and returns a string containing that information.

```
function get_person_info(name, age)
return name .. " is " .. age .. " years old."
end

print(get_person_info("John", 25)) -- Outputs "John is 25 years old."
```

Lua also has support for several other data structures, including user data, which is used to store arbitrary data, and threads, which are used to implement concurrent programming.

Multiple functions can be wrapped inside a module. Modules must be in a different file and be in the same folder when called. To use a module and all functions inside, use require("module_name") and yes, quotes are needed around the module name.

```
–First create a table. Since we can store any data type inside tables we can also store functions
local calculator = {} –our calculator module
–Then, declare a function(s)
function calculator.add(a,b)
print(a+b)
end,
function calculator.subt(a,b)
print(a-b)
end,
function calculator.mul(a,b)
print(a*b)
end,
function calculator.div(a,b)
print(a/b)
print(a/b)
–etc. as many function as you like, make sure to return calculator
return calculator

–accessing the calculator module
local calc = require("calculator")
```

calc.add(1,2) –Output: 3
calc.subt(1,2) –Ouptut: -1
calc.mul(1,2) –Output: 2
calc.div(1,2) –Output: 0.5

Functions can also be anonymous which are called inside a local or global function. These functions are known to also be closures since they close over local variables. Local variables above the function can still be accessed by the anonymous function. Furthermore, these variables above the anonymous function are called upvalues since they lay above the anonymous function. For example:

```
local function counter()
local i = 1
return function() –closure function(anonymous)
i = i + 1
return i
end
end

–Demo below shows how values are saved specifically to variable names
local v1 = counter()
print("Value of v1: ", v1()) –Output:  2
local v2 = counter()
print("Value of v1: ", v1()) –Value is saved Output: 3
print("Value of v2: ", v2()) –Output: 2
```

As shown above, invoking counter() starts adding i and saves the value to the variable from which it was invoked. This value sticks with the previous value v1 and continues to add to it when invoked again with a different variable v2. As shown, v1() outputs i = 3 since counter() was called twice but v2 outputs 2 despite the function being called twice because Lua interprets a new variable v2 as a new function call that starts at i = 1.

An extension to tables is metatables which are auxiliary tables that help modify the behaviors of tables. Metatables can be used to look up key-value pairs and by using metamethods such as ***__add, __sub, __mul, __div, __mod, __concat***, which are methods that change the primitive operators to work with tables. In order to set a metatable, one must use:
setmetatable(table_name, metatable) to set the metatable and
getmetatable(table) which gets the metadata of a table

Now that we know how to set up anonymous functions and how they can act as closures to access variables above them which are called upvalues, and tables, we can create our first metatable shown below:

```
–How to set a table as a metatable and check table for index
local mytable = {1, 2, 3, 4, 5}
setmetatable(mytable, {
–Index checks if there is a key that exists in the table
__index = function(atable, index) —anon function which is then stored in __index
return “No such data exists” –prevents table search out of bounds index from crashing
end
})
print(mytable[100], “\n”) –Output: No such data exists
```

Using meta tables can also modify tables to have more indexes from when it was originally set to.

```
setmetatable(mytable, {
__newindex = function(atable, index, value)
–rawset 3 parameters, atable which fetches mytable, index, and the value of the index
rawset(atable,index,value)
print(“added ”, value, “to mytable at index ”, index, “\n”)
return 1 –prevents table search out of bounds index from crashing
```

end
})
mytable[6] = 1 —Output: added 1 to my table at index 6
print(mytable[6], "\n") –Output: 1

Simply by typing an index outside of mytable (mytable[6] = 1) invokes the metatable to come in and handle the error.

We can also see that the metatable has a different address in memory compared to mytable. This is to show that even though we are setting up the metatable and table together, they are still located in different addresses shown below:

print(mytable) –Output: table: 0000000000e62f80
print(getmetatable(mytable)) –Output: table: 0000000000e63200

To change the behavior of the addition function we must call it inside a metatable. Just like the programming APL developed in the 1960s by Kenneth E. Iverson, we can add tables values together using the __add function.

local mytable = setmetatable({2,4,6}, {
__add = function(mytable, anothertable)
–# gets the length of mytable as well as string length. Lua handles the rest and outputs the
–length according to the data type.
for i = 1, #mytable do
table.insert(mytable, #mytable+1, newtable[i])
end
return mytable
end
})
local mytable2 = {4,5,6}
mytable = mytable + mytable2

for i = 1, #mytable, 1 do
print(mytable[i]) –Output: 1,2,3,4,5,6
end

Because the metatable functionality is set, we can now combine any two arrays together by using
mytable = mytable + ntable where n can be any number of the tables created.

Now with the power of metatables, tables, and functions, OOP functionality can be mimicked. The reason why it is mimicked is that it is not an actual class but a prototype. A prototype is a regular object where the first object looks up an operation that it does not know about yet by using the __index metamethod we used above. Classes are different because they are syntactically simplified and also the objects that are created through the classes are reliant on the classes since the classes provide the blueprint to create the object such as the Java language. Prototypes are not reliant on classes and you can create objects without the need for a class and the compiler will not throw an error. For example:

–First, we define Rectangle which consists of 3 key-value pairs
–As you can see, we already made a Rectangle object through a table without a blueprint.
Rectangle = {area = 0, length = 0, width = 0}

–However, we can only use this for this Rectangle. We must create a metatable and
--metamethods to assign new variables with different area, length, and widths

—Method constructor use colin to assign a function called new() derived from Rectangle
function Rectangle:new(o, length, width)

o= o or {} –If there is no area assign it a blank table if there is assign to to current area
setmetatable(o, self)
self.__index = self

```
self.length = length or 0
self.width = width or 0
self.area = length * width
return area
end

function Rectangle:printArea() –Assigning Rectangle another functionality
print("The area of Rectangle is ", self.area)

–Create a object
local rectangle1 = Rectangle:new(nil, 10, 10) –Output: The area of the Rectangle is 400
local rectangle2 = Rectangle:new(nil, 20, 20) –Output: The area of the Rectangle is 400.
```

Furthermore, only one object can be created at a time as both outputs produce the final function call which is 400 and true classes can create multiple objects at a time that are uncoupled and produce different outputs.

Recursion can be implemented in Lua by defining the function name and then calling the same function name within itself which will repeat until a base case is reached.

```
local function factorial(n)
if n == 0 then
return 1
end
return n * factorial(n-1)
end
print(factorial(5)) –Output: 120
```

Use Cases

Lua is used highly in Roblox, an online website founded by the David Baszucki, the current CEO of Roblox(2022), and Erik Cassel. In Roblox, players can customize their characters, play together with millions of people across the world, and players can also build their own games using Lua. However, Roblox's version of Lua is known as Luau or formerly RBX.lua, it still has all the features of Lua with performance optimization, gradual typing which is typing dynamically or statically, and a plethora of functions that are Roblox-specific to make a working game! Luau can be thought of as a superset of plain Lua just like the relationship between C++ and C where C++ is the superset of C.

In game development, players can choose to couple their scripts with objects and manipulate values inside their scripts to manipulate the objects, attributes, and behaviors. In foresight, this is a bad idea. Lua created modularity in order to decouple their code and use the DRY(Do not repeat yourself principle). If a developer decides to copy and paste the same Lua files into more objects that need the same functionality, and another developer must change the functionality, that other developer will have to go into every single file leading to more work, more time wasted, and loss of profit. When we have modularity, we can simply require the function, and then insert it into files as needed. Another solution is to use another Lua file to control what Lua files go into which object. Command arguments are a lot faster to implement than graphical user interfaces. Finally, coroutines should be implemented when there is a multi-functionality of one object. Games need to load in multiple Lua files and there is a high chance separate scripts will not load at the exact same time which can offset the timing of the desired functionality. By using coroutines, multi-functions can coexist in one file and the developer has more control over the timing of function firing to get the desired output. (*Sources come from CS431 Slides 99-105 that explain in depth with pictorials and a video demonstration. Link on page 27)*

In conclusion, Lua provides a variety of data structures and OOP that can be used to effectively store and manipulate data. From the basic table to metatables and primitive data types to more complex structures like functions and threads, Lua offers a wide range of options for working with data.

Tuckers Criteria

Tucker's criteria is a standard way of reviewing or rating how good a programming language is. There are many factors that go into this criterion including expressivity, well-definedness, data types and structures, modularity, I/O facilities, portability, efficiency, pedagogy, and generality.

Expressivity in the Lua programming language is very good as the variable names are fairly unrestricted which means variables can be given descriptive names. Mathematical operations follow familiar rules and simple precedence so it allows for all kinds of calculations. Lua also has a plentiful amount of control structures for users to implement. These structures include if then, for, else, and while loops that help with program design and functionality.

The well-definedness of Lua is also good, however, functions are anonymous so it can be hard to follow at times. Also, tables use key-value pairs which can be confusing at first use. There are many properties to the data type tables in Lua and can take a while to fully learn how to effectively implement them.

Lua has a variety of useful data types and structures allowing users to complete almost any task they may need to. Tables are a huge part of Lua's data types because you can store any value or use any value in Lua as a key in a table. Tables can also be used to implement associative arrays which means that any data type supported by Lua can be used to index an array.

Lua's modularity is very good as it was designed to be embedded with other languages. It was made to be very modular from the beginning which is what makes Lua such a lightweight and portable language. Lua is often even referred to as a "glue language", meaning it is used to help tie coding languages together. Within Lua, modularity is easy to implement due to powerful control structures, both built-in and the ability to define custom functions, and object-like functionality through the use of tables. Lua can call functions defined in Lua and also in C, or

whatever language happens to be the hosting environment. Recursive function calling is enabled by default, so even repetitive tasks can be broken down into smaller tasks as much as necessary.

The I/O facilities in Lua are also very good. Lua offers fairly precise control over input and output channels through two different models: the simple model and the complete model. The simple model is the most straightforward and easy to use, allowing the programmer to read from and write to a separate "current" file for each operation. The current files are set to the standard input and standard output by default and will continue referencing these files until they are explicitly changed with calls to io.input(filename) orio.output(filename) respectively. The complete model allows more fine-tuned control over input and output.

Lua is a very portable language as it was originally designed to be. This language was meant to be small, fast, and easily implemented into applications. Lua is distributed in a small package and builds out-of-the-box on all platforms with a standard C compiler. Lua is able to run on all different kinds of Unix, macOS, Windows, and mobile devices and can even be optimized for embedded systems. Lua's core package at only 1.3 MB is small enough to be implemented on even the most resource-constrained systems, and the fact that it is open source means that there are no artificial, licensing obstacles to its use throughout the programming world.

The efficiency of Lua is good but not great. Lua is an interpreted language and is not as efficient as a compiled language, such as java, would be. Although the efficiency of Lua is not great, it has good efficiency for a scripting language. It is much faster than other scripting languages such as python or rust. Lua is probably more efficient than some other languages due to its intended purpose: to be embeddable across as many platforms as possible regardless of system restrictions like limited memory and processing power. In fact, Lua generally leaves as much on the table as possible when it comes to functionality that is not absolutely necessary. It does this partly by keeping its core implementation small and relying on the host language (usually C) for the heavy lifting of computationally intensive tasks. This is not to say that Lua is incapable of accomplishing these tasks, but it does not attempt to reproduce what other languages are already good at. Lua is concise and to the point.

Learning a new programming language can be a daunting task whether it is the first language one is studying, or if one already has several languages under their belt. However, understanding the similarities and differences between languages provides some grounding and a reference point to build off of when previous programming experience is present. This lessens the slope of the learning curve and can drastically lessen the amount of time needed to become proficient in the new language. As such, it can be difficult to honestly assess the pedagogy of a language when several languages are already fairly well understood. With that being said, Lua seems to be an easy language to learn - at least in its simplest form. Some of the features that make learning Lua less painful than other similar languages are; the fact that variables are dynamically typed, the interpreter has stand-alone functionality, and debugging does provide some - if incomplete - error descriptions. Not everything in Lua is so straightforward though. Tables are complex objects conceptually and unleashing their full potential does require some dedicated attention. Data structures such as arrays in other languages are perhaps a simpler idea to get your head around, and while Lua does provide array-like functionality through the use of its tables, it may not be the simplest introduction to the concept. All in all, Lua would not be a bad place to start learning to program, and certainly would not be a problematic addition to an experienced programmer's repertoire.

As a general programming language, Lua is a good choice on many fronts. It is well established as an object oriented programming language, a procedural programming language, and a functional programming language to name a few. It can be used as a general language for small scripting type tasks as well as being embedded with other languages for the creation of large complex projects being deployed over various networks and platforms. Lua is compatible with virtually all operating systems and requires only small customizations to its build settings to achieve this compatibility. Often Lua is referred to as a glue language due to its usefulness in "gluing together" other components to produce efficient and easy to follow program design. When used in this way, Lua's scripting characteristic allows the programmer to make basic design changes to their program without recompiling code. This along with the other reasons above make Lua a great choice for scholarly, experimental, and real-world applications.

References

Roberto Ierusalimschy, Lua.org, December 2003, Programming in Lua (first edition), Accessed December 2022,
https://www.lua.org/pil/contents.html

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, Copyright © 2020–2022 Lua.org, PUC-Rio, Lua 5.4 Reference Manual, Accessed December 2022,
https://www.lua.org/manual/5.4/manual.html

Lua Download Sources and Instructions
https://www.lua.org/download.html

Lua.org
http://www.lua.org/

Lua use cases
https://www.tutorialspoint.com/lua/lua_game_programing.htm

Luau Roblox
https://roblox.fandom.com/wiki/Lua

CS431 Lua slides 99-105
https://docs.google.com/presentation/d/159SfdWHnpJ849H772bxsjwIbsSuUMwvUw-IrwhZSfG0/edit?usp=sharing