
LUA

Yixin Li
Computer Science
University of Arizona
carolineliyixin@email.arizona.edu

Jiahui Wang
Computer Science
University of Arizona
wangj2@email.arizona.edu

April 29, 2020

ABSTRACT

1 Introduction

Lua is a small, simple, fast yet powerful and successful language that develops full functionalities and wild extensibilities yet easy to implement and debug. Lua was purposely designed to be integrated into other conventional languages, which make itself relies on wild extensibilities that may depend either on code written in Lua itself but also in other languages like C/C++, and so it is wise for Lua to implement features that complement with C/C++ to achieve a better overall performance. Lua is also a "glue" language that gluing together the existing high-level components.

One of the reasons that Lua is so simple but still powerful is Lua's use of libraries. One of the representatives of this feature is that Lua use "table" as its only data structure that acts as an associative array that can store key-value pairs. It plays a central role in implementing arrays, modules, objects, sets, and many other data structures. Lua supports dynamic typing that allows polymorphism; it also supports automatic memory management, higher-order functions, and anonymous functions, making Lua functions diverse.

2 History

Lua is a scripting language that was first developed at the Pontificia Catholic University of Rio De Janerio in Brazil in 1993. Lua was created because back in those days when computer languages were not very flexible. Many large machines need to be operated based on a large amount of data, which was dependent on the input by individual programmers. This is very demanding on the programmer because a small mistake can have a great impact on the entire program, and thus Lua was proposed To simplify and to improve the safety of the program. Many machine languages use a top-down structure, which is not conducive for detail modification. Lua was created to make it simpler and faster to modify the entire program.

Lua has helped many large programs with its simplicity and portability. The development of Lua has far exceeded the most optimistic expectations, very likely because of its design principle where simplicity was enforced. Lua has been continuously modified and updated to be widely used. Besides being a leading scripting language in gaming, it has all kinds of industrial applications including robotics, image processing, literate programming, and so on.

3 Control Structures

The control structure in Lua is fairly straight forward and common as in many popular languages. There are two types of control structure: conditional structure includes "if-then-else", and iteration structure includes for, while, repeat for .

All the control structure ends with a key word "end", except that *repeat*, we need no "end" for the iteration but with a key word *until*. One thing to notice is that Lua treats all values beside false and nil to be true.

The only conditional control structure is *if-then-else*, and there are several ways to use it:

```

1 --example 1
2 if a > 0 then b = 0 end
3
4 --example 2
5 if a > 0 then return a else return b end
6
7 --example 3
8 n = 5
9 if n > 5 then
10   print("greater than 5")
11 elseif n == 5 then
12   print("equal to 5")
13 else
14   print("less than 5")
15 end

```

Listing 1: if-then-else

The key word here are **if**, **elseif**, **then**, and **else**. All key words have fairly unambiguous meaning, except that **then**, which just indicates the sentence after **then** is a "statement", where some actions may be followed by if the condition before "then" is evaluated to true.

There are two types of for loop, the *numeric for* loop and *generic for* loop, which are shown respectively in example 1 and example 2:

```

1 --example 1: numeric for
2                                     --syntax
3 for i = 0, 10, 2 do                 --for var=exp1,exp2,exp3 do
4   print("hello world")             -- statement
5 end                                 --end
6
7 --example 2: generic for
8 days = {"Sunday", "Monday", "Tuesday", "Wednesday",
9         "Thursday", "Friday", "Saturday"}
10
11 for i,v in pairs(days) do
12   print(i,v)
13 end

```

Listing 2: for loop

Numeric for loop looks a lot like for loop in python. For example 1, local variable *i* start at 0, and end at 10, with 2 increment on *i* after each iteration. The third expression here is optional as the default increment value is 1. The keyword **do** separate the for loop condition and the statement, and the loop end with **end**. Generic for loop can be used for even a wilder application, especially in looping through key-value pairs using **pairs** or **ipairs** from a table, or reading from a file (using `io.lines`). For example 2, *i,v* represent the index and value pair in the table (similar to a dictionary in python). In this case, will print out

```

1 0 Sunday
2 1 Monday
3 .
4 .
5 6 Saturday

```

The structure of the while loop is very similar to many languages, as shown in the example bellow.

```

1 i = 0
2 while i < 10 do
3   print("hello world")
4   i += 1
5 end

```

Listing 3: while loop

The repeat-until is very similar to the while loop. Except what while loop begins with evaluate the condition, and only proceed into the statements if the condition first evaluates to true, while repeat-until starts with proceed the statement before evaluating a condition and decide if proceed again. In other words, repeat-until will iterate at least once.

```

1 i = 10
2 repeat
3   print(i)
4   i = i - 1
5 until i == 0

```

Listing 4: repeat-until

The example above will print from 10 to 1 line by line.

4 Data Types

Lua is a dynamically typed language. Variables do not need to define the type before the assignment. However, a value type can be stored in a variable and pass as a parameter or as a return value.

There are eight data types in the Lua. These include nil, boolean, number, string, table, function, user data, and thread. We can use type() to check the type of value.

nil is one of the data types that is used to differentiate the value from having some data or no(nil) data. For example, the result of printing a type of variable that does not assign any value is nil. When nil type is used on compare operation, it needs to be used as "nil", otherwise, it will be recognized as a string. Nil also represents false in the Lua.

```

1 -- n doesn't assign value
2 -- the result is nil
3 print(type(n))
4
5 -- compare operation
6 print(type(n) == "nil") -- true

```

Listing 5: nil

The **boolean** type is used to perform condition checking. It only has two values which are true and false. In Lua, only false and nil represent **false**, and any other are **true** include 0.

Boolean type is used to evaluate conditions in the if-then-else data structure or some other compare operations.

```

1 -- the type of boolean
2 print(type(false))
3 print(type(true))
4
5 -- the operation
6 if false then
7   print("false")
8 else
9   print("true")
10 end
11 -- nil is the same as false in lua
12 if nil then
13   print("false")
14 else
15   print("true")
16 end
17 -- 0 is the same as true in lua
18 if 0 then
19   print("true")
20 else
21   print("false")
22 end

```

Listing 6: boolean

The type **number** represent real numbers. It is only one digit type. Integer, float and some other numeric types are all regarded as numbers. We can directly use it to perform calculations.

```

1 -- there is only number type in lua
2 -- all the result is number
3 print(type(6))
4 print(type(6.6))
5 print(type(6e + 1))

```

Listing 7: number

The **string** type is used to represent the array of characters in Lua. It is recognized by `""` or `''`. These two expression forms are the same, which is different from java.

When we do the digit calculate operations with digit string such as `"1"`, Lua attempt to transfer string to number and then execute the operation. This is much simpler than java and some other languages. However, if one uses a string like `"hello"` to add a number, Lua can not transfer it and will cause the error.

The operator to connect the strings is `..`, while `""` is used to compute the length of the string.

```

1 -- "" and '' both can represent string
2 str1 = "hello"
3 str2 = 'world'
4 print(str1)
5 print(str2)
6
7 -- operation for digit operate
8 print("3" + 4)
9
10 -- operation for connect the string
11 print("a" .. "b")
12 print(123 .. 456)
13
14 -- operation for compute the string length
15 print(#str1)

```

Listing 8: string

The **table** is one of the most critical data structures in Lua. It has diverse functionalities that can represent arrays, symbol tables, sets, graphs, trees, etc. It implements associative arrays, and the index of the array can be either a number or a string.

There are two ways to create a table. The first one is to directly write all the elements in the table. The other way is to create an empty table first and then use the built-in functions to add elements in the table.

We can change the table content by either directly change one of the elements or use built-in functions. If we change the value to `nil`, this will be the same as deleting this key and value from the table.

```

1 -- create a table
2 local table1 = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"}
3
4 -- Traverse the table and print it
5 for key, val in pairs(table1) do
6     print(key .. ":" .. val)
7 end
8
9 -- operation for table
10 -- insert
11 table.insert(table1, 1, "Sunday")
12
13 -- delete
14 table.remove(table1, 2)
15
16 -- sort the table
17 table.sort(table1)
18
19 -- Print with specified situation
20 print(table.concat(table1, ",", 1, 5))

```

Listing 9: table

The **function** is also a data structure in Lua. This is the same as the method in Java, which used to achieve the goal of the programmer.

```

1 --function compute the sum of even number for 0 to the given number
2
3 function sum_even(n)
4     if n == 0 then
5         return n
6     else
7         if n % 2 == 0 then
8             return n + sum_even(n-1)
9         else
10            return sum_even(n-1)
11        end
12    end
13 end

```

Listing 10: function

thread represent the independent threads of execution and is used to implement co-routines in Lua. Coroutine in Lua is mostly the same as thread as it has its own independent stack, local variables, and instruction pointers, and is able to share global variables and many other things with other cooperating programs. The difference between thread and coroutine is a thread can run multiple times at the same time, but a coroutine can only run once at one time, and the only way to the temporary stop the coroutine during a running state is changed to the suspend state.

The **userdata** is user-defined data to represent arbitrary C/C++ library and store the data in the Lua.

5 Subprograms

Many languages provide mechanisms to organize space and allocate different components of the project, such as *packages* for Java, *namespaces* in C++, in order to provide a mechanism to avoid collisions among names among different part (libraries) of the project, especially for projects that built enormously.

Lua does not build any special mechanism to perform the functions of a normal package. However, Lua stores **all** global variables for each file in regular Lua tables, called **environment**. Users will be able to access the functions from other files as well as functions in libraries by accessing the global variables and functions by simply use “require” keyword to “import” by the package name.

Lua creates a table for the environment itself, called `_G`, so one would be able to access all packages by setting the current (use a local variable) environment equal to `_G`. However, it is more decent to only declare local variables as need, which makes the code more clear and better at performance.

```

1 -- A concise solution
2 local P = {}
3 pack = P
4 local _G = _G -- Access the global environment
5 setfenv(1, P)
6
7 -- A better solution
8 local P = {}
9 pack = P
10
11 -- Import Section:
12 -- Only declare what this package needs from outside
13 local sqrt = math.sqrt
14 local io = io
15
16 setfenv(1, P) -- no more external access after this point

```

Listing 11: Lua environment and access packages

The relationship between file and package is interesting. People write their code for building packages inside a file, and when the package is needed to be available, we will simply run the file, or import by `P = require(filename)`;

```

1 -- In file named a.lua, we implement a simple function

```

```

2 M = {}
3 function M.add10(a) return (a + 10) end
4 return M -- This return is not necessary
5
6 -- Now in file b.lua, we want to introduce the function
7 -- add10 written in package M in a.lua
8 M_a = require("a")
9 print(M_a.add10(10)) -- will print 20
10
11 -- Note that require key word depend on file name,
12 -- NOT package Name.

```

Listing 12: write and use packages

There are many valid ways to write packages in Lua

```

1 -- In file a:
2 -- We store all of the functions
3 -- into a table named operations, and
4 -- return the table in the end.
5 local operations = {}
6
7 function operations.add(a,b) print(a + b) end
8 function operations.sub(a,b) print(a - b) end
9 function operations.mul(a,b) print(a * b) end
10 function operations.div(a,b) print(a / b) end
11
12 return operations
13
14 -- In file b:
15 -- pass the table of functions from
16 -- file a and use them in file b.
17 operator = require("operations")
18 operator.add(10,20)
19 operator.sub(30,20)
20 operator.mul(10,20)
21 operator.div(30,20)

```

Listing 13: The basic approach

In many circumstances, "private" functions are needed under both decoration and functional needs, as what we describe above may give a wrong illusion that Lua makes every function public to any client who has access to the name of the file. There are indeed elegant ways of writing a "public" or "private" functions in Lua. Although writing a mixture of public and private functions in Lua is perfectly fine, the example code shown below illustrate a better way of writing package by keeping all functions private and return the global variable that contains the functions we do want to make public of in the end.

```

1 local function checkrange(a,b,n)
2   if not n > b and n < a then error("Not in range") end
3 end
4
5 local function new(a)
6   return {a = a}
7 end
8
9 local function check(a,b,n)
10  checkrange(a,b,n);
11  return new(n)
12 end
13
14 complex = {
15   new = new
16   check = check
17 }

```

Listing 14: function

There are many benefits to implement the package like the way in Lua, as it is very easy to manipulate packages and even create extra facilities.

The feature of table in Lua also makes it possible for this language to have Object-Oriented(OO) programming features. One table is able to represent an "Object" because it can store state and function independently to any other tables that have the same attributes to itself. One table will also have its own life circle depend on when and where it is initiated.

Just like all OOP languages, the need to use self is a critical point that achieves the creation of independent "Objects" that may or may not need to possess same attributes and functionalities.

```

1 BankAccount = {}
2 -- Without use of colon
3 function BankAccount.deposit (self, n)
4     self.balance = self.balance + n
5 end
6 -- WITH use of colon
7 function BankAccount:withdraw (n)
8     self.balance = self.balance - n
9 end
10
11 -- Without use of colon
12 BankAccount.deposit(Account, 200.00)
13 -- WITH use of colon
14 BankAccount:withdraw(100.00)
15
16 -- Another object a shares the same attributes and functions as BankAccount.
17 a = BankAccount
18 a:deposit(100.00)
19 a:withdraw(a, 100.00)
20 -- The use of colon abbreviated the need to repetitively write "self", as the use
    of self is really a central point in almost very OO-design programming
    languages.

```

Listing 15: OOP

6 Summary

Lua is a language that aims to be simple to implement, fast to compile, and perform full functionalities. Specifically, Lua has an important feature that uses "table" to act as an associative array that can store key-value pairs. It plays a central role in implementing arrays, modules, objects, sets, and many other data structures. Lua supports dynamic typing that allows polymorphism; it also supports automatic memory management, higher-order functions, and anonymous functions, making Lua functions diverse.

7 References

References

- [1] Roberto Ierusalimschy (2016) *TEX: Programming in Lua*, Roberto Ierusalimschy, 4th ed.
- [2] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes (2007) *TEX: The evolution of Lua*, Proceedings of the third ACM SIGPLAN conference on History of programming languages.
- [3] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes (2006) *TEX: Lua5.1 reference manual*.
- [4] *Lua community*. <https://www.reddit.com/r/lua/>
- [5] *Lua projects*. <http://lua-users.org/wiki/LuaProjects>