LUA

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

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

April 19, 2020

ABSTRACT

The abstract of the paper

1 Introduction

2 History

Lua is a scripting language that was first developed at the Pontificla Catholic University of Rio De Janerio in Brazil. Lua was created because back in those days when computer languages were not very flexible. Many large machines needs to be operated where running a large amount of data 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. To simplify and to improve the safety of the program, Lua was proposed. 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 languages 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: ”if-then-else” for conditional and for, while, repeat for iteration.

All the control structure ends with a key word ”end”, except that for 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*, and there are several ways to use it:

```
--example 1
if a > 0 then b = 0 end

--example 2
if a > 0 then return a else return b end

--example 3
n = 5
if n > 5 then
  print("greater than 5")
elseif n == 5 then
  print ("equal to 5")
```

```
else
  print("less than 5")
end
```

Listing 1: if-then-else

The key word here are **if**,**elseif**, **then**, **else**. All key words here have fairly unambiguous meaning, except that **then**, which just indicate the sentence after it is the "statement", where some actions may be following if the condition before "then" is evaluate to true.

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

```
--example 1: numeric for
                                --syntax
for i = 0, 10, 2 do             --for var=exp1,exp2,exp3 do
  print("hello world")          --   statement
end                             --end

--example 2: generic for
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"}

for i,v in pairs(days) do
  print(i,v)
end
```

Listing 2: for loop

Numeric for loop looks a lot like for loop in python. In 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 key word **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 applications, especially in looping through key-value pairs using **pairs** or **ipairs** from a table, or reading from a file (using io.lines). In example 2, i,v represent index and value of a pair in the table (similar to a dictionary in python). In this case, will print out

```
0 Sunday
1 Monday
.
.
6 Saturday
```

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

```
i = 0
while i < 10 do
  print("hello world")
  i += 1
end
```

Listing 3: while loop

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

```
i = 10
repeat
  print(i)
  i = i - 1
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 define the type before 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, userdata and thread. We can use type() to check the type of value.

nil is the one of the data types that is used to differentiate the value from having some data or no(nil) data. For example, print a type of variable which not assignment any value, the result is nil. When nil type use on compare operation, need use as "nil", otherwise, it will recognize as string. It also represent false in the Lua.

```
-- n doesn't assign value
-- the result is nil
print(type(n))

-- compare operation
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 a evaluate condition in the if-then-else data structure or some other compare operations.

```
-- the type of boolean
print(type(false))
print(type(true))

-- the operation
if false then
  print("false")
else
  print("true")
end
-- nil is the same as false in lua
if nil then
  print("false")
else
  print("true")
end
-- 0 is the same as true in lua
if 0 then
  print("true")
else
  print("false")
end
```

Listing 6: boolean

The **number** type in Lua 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.

```
-- there is only number type in lua
-- all the result is number
print(type(6))
print(type(6.6))
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 form are the same, which is different from java.

When we do the digit calculate operations with digit string such as "1", Lua attempt 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.

```
-- "" and '' both can represent string
str1 = "hello"
str2 = 'world'
print(str1)
print(str2)

-- operation for digit operate
print("3" + 4)

-- operation for connect the string
print("a" .. "b")
print(123 .. 456)

-- operation for compute the string length
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. First one is directly to write all the elements in the table. The other is create 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 element or use built-in functions. If we change the value to nil, this will be same as delete this key and value from the table.

```
-- create a table
local table1 = {"Monday","Tuesday", "Wednesday", "Thursday", "Friday"}

-- Traverse the table and print it
for key,val in pairs(table1) do
  print(key .. ":" .. val)
end

-- operation for table
-- insert
table.insert(table1,1,"Sunday")

-- delete
table.remove(table1,2)

-- sort the table
table.sort(table1)

-- Print with specified situation
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.

```
--function compute the sum of even number for 0 to the given number

function sum_even(n)
  if n == 0 then
    return n
  else
    if n % 2 == 0 then
      return n + sum_even(n-1)
    else
      return sum_even(n-1)
    end
```

```
  end
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 same as thread as it has 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 change 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 mechanism to organize space and allocate different components of the project, such as *packages* for Java, *namespaces* in C++, in order to provide 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 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" key word to "import" by the package name.

Lua creates an table for 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.

```
-- A concise solution
local P = {}
pack = P
local _G = _G -- Access the global environment
setfenv(1, P)

-- A better solution
local P = {}
pack = P

-- Import Section:
-- Only declare what this package needs from outside
local sqrt = math.sqrt
local io = io

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);

```
-- In file named a.lua, we implement a simple function
M = {}
function M.add10(a) return (a + 10) end
return M -- This return is not necessary

-- Now in file b.lua, we want to introduce the function
-- add10 written in package M in a.lua
M_a = require("a")
print(M_a.add10(10))  -- will print 20

-- Note that require key word depend on file name,
-- NOT package Name.
```

Listing 12: write and use packages

There are many valid ways to write packages in Lua

```
-- In file a:
-- We store all of the functions
-- into a table named operations, and
-- return the table in the end.
local operations =  {}

function operations.add(a,b) print(a + b) end
function operations.sub(a,b) print(a - b) end
function operations.mul(a,b) print(a * b) end
function operations.div(a,b) print(a / b) end

return operations

-- In file b:
-- pass the table of functions from
-- file a and use them in file b.
operator = require("operations")
operator.add(10,20)
operator.sub(30,20)
operator.mul(10,20)
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.

```
local function checkrange(a,b,n)
  if not n > b and n < a then error("Not in range") end
end

local function new(a)
  return {a = a}
end

local function check(a,b,n)
  checkrange(a,b,n);
  return new(n)
end

complex = {
  new = new
  check = check
}
```

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.

```
BankAccount = {}
-- Without use of colon
```

```lua
function BankAccount.deposite (self, n)
  self.balance = self.balance + n
end
-- WITH use of colon
function BankAccount:withdraw (n)
  self.balance = self.balance - n
end

-- Without use of colon
BankAccount.deposit(Account, 200.00)
-- WITH use of colon
BankAccount:withdraw(100.00)

-- Another object a shares the same attributes and functions as BankAccount.
a = BankAccount
a:deposit(100.00)
a.withdraw(a, 100.00)
-- 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

7 References

References

[1] Roberto Ierusalimschy (2016) *LaTeX: Programming in Lua*, Roberto Ierusalimschy, 4th ed.

[2] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes (2007) *LaTeX: 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) *LaTeX: Lua5.1 reference manual*.

[4] *Lua community*. https://www.reddit.com/r/lua/

[5] *Lua projects*. http://lua-users.org/wiki/LuaProjects