LUA

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

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

April 14, 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, the value has types, it can be stored in an variable and pass as parameters or as return value.

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

The **nil** is the one of the data type that use 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 assignment value
-- the result is nil
print(type(n))

-- compare operation
-- the result is true
print(type(n) == "nil")
```

Listing 5: nil

The **boolean** type is use to represent the condition checking. It only has two value that are true and false. In Lua, false and nil represent **false**, and the others are **true** include the value 0.

Boolean type can use as a 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.The integer, float and some other types all be regarded as number. 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 use to represent the array of characters in the Lua. It recognized by "" and ". These two expression form are the same. This is different with java.

When we do the digit calculate operations with digit string such as ”1”, Lua attempt transfer the string to number and execute the operation. This is much simplier than java and some other languages. However, if use 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 .. and ”” is to compute the length of the string.

```lua
-- "" 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 represent arrays, symbol tables, sets, graphs, trees, etc., it implements associative arrays,the index of the array can be a number or a string.

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

For change the table content can directly to change one of the element, or also use built-in functions. If change the value to nil, this is the same as delete this key and value.

```lua
-- 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
-- instert
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 represent the method which written in Lua. This is the same as the method in Java. Use to achieve the goal which user want.

```lua
--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

The **thread** is represent the independent threads of execution and it is used to implement coroutines in Lua. Coroutine in Lua is mostly same as thread it has own independent stack, local variables and instruction pointers, can share global variables and most other things with other cooperating programs. The different between thread and coroutine is a thread can run multiple times at the same time, but a coroutine can only run one at any time, and the only way to the temporary stop the coroutine in the running state is that change state to the suspend.

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

5 Subprograms

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