Lua versus Go

This article compares Lua and Go in respect of basic syntax, not including concurrent programming paradigm. It is based on Learn Lua in 15 Minutes. In the following text, the left column shows Lua code and the right column shows coresponding Go code.

Variables and Flow Control

Variable definitions and `nil` value.

```
num = 42
s = 'walternate'
t = "double-quotes are also fine"
u = [[Long long
lines]]
t = nil
```

```
num := 42
s := "walternate"
t := "single-quotes are not allowed"
u := `Long long
lines`
t := nil
```

In Lua, blocks are denoted with keywords like do/end. In Go, they are curly parenthesis.

```
while num < 50 do
    num = num + 1
end
```

```
for num < 50 {
  num++
}
```

If clauses, I/O, and string concatenation.

```
if num > 40 then
  print('over 40')
elseif s ~= 'walternate' then
  print('not over 40')
else
  thisIsGlobal = 5
  local line = io.read()
  print('Winter is coming, ' .. line)
end
```

```
if num > 40 {
  print("over 40")
} else if s != "walternate" {
  print("not over 40")
} else {
  thisIsGlobal = 5
  line := bufio.NewReader(os.Stdin).ReadString(`\n`)
  print("Winter is coming, " + line)
}
```

In Lua, undefined variable has value nil. In Go, undefined variable causes compiler error. In Lua, both false and nil are falsy; 0 and '' are true. In Go, only false is falsy.

```
foo = anUnknownVariable -- Now foo is nil.

aBoolValue = false
if not aBoolValue then print('twas false') end
```

```
foo = anUnknownVariable // compiler error here.

aBoolValue := false
if !aBoolValue { print("twas false") }
```

In both Lua and Go, 'or' and 'and' are short-cuircuited. In Lua, because only nil and false are falsy, all other values are true, so we can use 'and' and 'or' to do the a?b:c operation in C/Js. But in Go, there is nothing like a?b:c.

```
ans = aBoolValue and 'yes' or 'no'
```

```
if aBoolValue {
  ans = "yes"
} else {
  ans = "no"
}
```

Or, for Go, we can define a selection function.

```
func sel(cond bool, x, y interface{}) interface{} {
  if cond { return x }
  return y
```

```
}

ans := sel(true, "yes", "no").(string)
```

Lua provides a concise for loop syntax than Go. But really no much difference at use.

```
karlSum = 0
for i = 1, 100 do
  karlSum = karlSum + i
end

fredSum = 0
for j = 100, 1, -1 do fredSum = fredSum + j end
```

```
karlSum := 0
for i := 1; i <= 100; i++ {
  karlSum++
}

fredSum := 0
for j := 100; j <= 1; j-- { fredSum += j }
```

Functions

```
function fib(n)
  if n < 2 then return 1 end
  return fib(n - 2) + fib(n - 1)
end
```

```
func fib(n int) int {
  if n < 2 { return 1 }
  return fib(n - 2) + fib(n - 1)
}
```

A notable pitfall of Lua here is that, if we forget the second return statement, Lua interpreter wouldn't give any warning, and the call to `fib(2)` would simply return `nil`!

```
function fib(n)
  if n < 2 then return 1 end
end
print(fib(2))
```

```
func fib(n int) int {
  if n < 2 { return 1 }
  // Go compiler will complain for missing return here.
}
```

Closures and anonymous functions are ok.

```
function adder(x)
  return function (y) return x + y end
end
a1 = adder(9)
a2 = adder(36)
print(a1(16))  -- 25
print(a2(64))  -- 100
```

```
func adder(x int) func(int) int {
  return func(y int) { return x + y }
}
a1 := adder(9)
a2 := adder(36)
print(a1(16))  // 25
print(a2(64))  // 100
```

Both Lua and Go support assignment of multiple values. But Lua wouldn't warn if the number of assignees differs from the number of operands.

```
x, y, z = 1, 2, 3  -- no problem
x, y, z = 1, 2, 3, 4 -- 4 is thrown away
```

```
x, y, z = 1, 2, 3  // no problem
x, y, z = 1, 2, 3, 4 // compiler complains
```

Both Lua and Go functions can return multiple values. The difference is Go compiler checks strictly the matching of parameters and arguments, and the matching of assignees and return values.

```
function bar(a, b, c)
  return 1, 2, 3, 4, 5
end
x, y = bar('apple') -- a is 'apple', b and c are nil.
-- now x is 1, y is 2, values 3 .. 5 are discarded.
```

```
func bar(a, b, c int) (int, int) {
  return 1, 2
}
x, y := bar('apple', 'orange', 'banana')
```

Both Lua and Go support variadic functions.

```
function print (...)
  for i,v in ipairs(arg) do
    print(i, v)
  end
end
```

```
func print (arg ...interface{}) {
  for i,v := range arg {
    print(i, v)
  }
}
```

In both Lua and Go, functions are first-class.

```
function f(x) return x * x end
f = function (x) return x * x end
```

```
func f(x float64) float64 { return x * x }
f := func(x float64) float64 { return x * x }
```

In both Lua, function calls with only one parameter doesn't need parenthesis. This is not true for Go.

```
print 'hello'
```

```
print("hello")
```

Tables and Maps

Table is Lua's only compound data structure. They are hash-lookup dicts. In Go, there is type map, which is also a hash dict. In Lua, list is table with integer typed (index) keys. In Go, lists are saved in arrays or slices, which, as common types, can be accessed more efficiently.

```
t = { key1 = 'value', key2 = false }
```

```
t := map[string]interface{} { "key1" : "value", "key2" : false }
```

Lua is dynamically typed, so a table can have various typed keys and values. Go is strongly typed, but by using the special type `interface{}`, it supports multiple-typed maps.

```
u = {['@!#'] = 'qbert', [{}] = 1729, [6.28] = 'tau'}
print(u[6.28])  -- prints "tau"
```

```
u := map[interface{}]interface{}{
    "@!#":          "qbert",
    &map[int]int{}: 1729,
    6.28:           "tau",
}
print(u[6.28].(string))
```

Please be aware of using an empty table/map as the key. In Go, we cannot use `map[int]int{}` as the key -- the Go runtime will complain that a map is not hashable. Instead, we need to use a pointer to the map as the key. This might make you feel that Go is less convenient as Lua, but the truth is not. The truth is that above Lua code implicitly used pointer to table, instead of the table itself, as the key -- exactly the same way as the Go version. And the following Lua code returns `nil`, instead of 1729 as you might expect. Because the key used for indexing is not the same table object used to construct table `u`.

```
print(u[{}])
-- prints nil, but not 1729.
```

```
print(u[&map[int]int{}].(float64))
// runtime complains that interface is nil, not float64
```

A commonly used programming pattern is Lua is calling functions with only one parameter doesn't need parenthesis. When this only parameter is a table, it looks like the following. This pattern is used in Torch.

```
function h(x) print(x.key1) end
h{key1 = 'Sonmi'}

torch.Tensor{1,2,3}
```

Lua and Go uses similar syntax for enumerating key value pairs in a table/map.

```
for key, val in pairs(u) do  -- Table iteration.
  print(key, val)
end
```

```
for key, val := range u {
  print(key, val)
}
```

Also, similar syntax for enumerating lists/slices.

```
v = {'value1', 'value2', 1.21, 'gigawatts'}
for i = 1, #v do  -- #v is the size of v for lists.
  print(v[i])  -- Indices start at 1 !! SO CRAZY!
end
```

```
v := []interface{}{"value1", "value2", 1.21, "gigawatts"}
for i := range v {
  fmt.Println(reflect.ValueOf(v[i]))
}
```

Metatables and Metamethods

Lua table is often used as *namespace*, as we use method name (string typed) as the key, and the first-class method as the value. Such kind of namespaces, when assigned to a table variable using `setmetatable`, is called *metatable*. In Go, we don't attach methods to a variable; instead, we attach methods to types. Another notable difference here is that in Lua there are some pre-defined methods that correspond to operators. For example, if we overload `__add` in a metatable, we actually defines the action of operator `+` on the variable attached with the metatable.

```
f1 = { a = 1, b = 2 } -- represents the fraction a/b.
f2 = { a = 2, b = 3 }

metafraction = {}
function metafraction.__add(f1, f2)
  sum = {}
  sum.b = f1.b * f2.b
  sum.a = f1.a * f2.b + f2.a * f1.b
  return sum
end

setmetatable(f1, metafraction)
setmetatable(f2, metafraction)

s = f1 + f2  -- call __add(f1, f2) on f1's metatable
```

```
type frac struct{ a, b int }

func (f1 frac) add(f2 frac) frac {
  return frac{f1.a*f2.b + f1.b*f2.a, f1.b * f2.b}
}

f1 := frac{1, 2}
f2 := frac{2, 3}
s := f1.add(f2)
```

It is really not a good idea to attach methods to variables directly. The following line would fail, because s has no metatable.

```
t = s + s -- would fail
```

Class-like Tables and Inheritance

A more reasonable way to use metatable is with *class-like tables*. However, class-like tables really cost more lines of code than Go's class-like mechanism. It is also much less readable.

```
Dog = {} -- define a "class"

function Dog:new()  -- 'a:b' is the same as 'a.b(self,'
  newObj = {sound = 'woof'}
  self.__index = self
  return setmetatable(newObj, self) -- setmetatable returns its first arg.
end

function Dog:makeSound()
  print('I say ' .. self.sound)
end

mrDog = Dog:new()
mrDog:makeSound()
```

```
type Dog struct {
  sound string
}

func NewDog() *Dog {
  return &Dog{"woof"}
}

func (d *Dog) makeSound() {
  fmt.Println("I say " + d.sound)
}
```

The Lua way of inheritance is also variables-based, and differs from Go's typed-based inheritance.

```
LoudDog = Dog:new()

function LoudDog:new()
  newObj = {}
  self.__index = self
  return setmetatable(newObj, self)
end

function LoudDog:makeSound()
  print(self.sound .. self.sound .. self.sound)
end

seymour = LoudDog:new()
```

```
type LoudDog struct {
  *Dog
}

func NewLoudDog() *LoudDog {
  return &LoudDog{NewDog()}
}

func (l *LoudDog) makeSound() {
  fmt.Println(l.sound, l.sound, l.sound)
}

seymour = NewLoudDog()
```

```
seymour:makeSound()
```

```
seymour.makeSound()
```

Modules and Packages

Lua's module definition and loading mechanism is similar to Javascript. It is an easy-to-implement design for interpreted languages -- the standard `require` function inserts the content of a moudule file, which is simply a Lua source file, into an anonymous function definition, and run that function. So the last statement of a Lua module source file is often a `return` statement. To install and maintain Lua modules, we need 3rd party tools like `luarocks`. However, Go provides a well designed packaging mechanism, which doesn't need a module/package management system like `luarocks` for Lua, `npm` for Javascript, or `Maven` for Java.

```
-- Suppose the file mod.lua looks like this:
local M = {}

local function sayMyName()
  print('Hrunkner')
end

function M.sayHello()
  print('Why hello there')
  sayMyName()
end

return M
```

```
package mod

// Only functions with capitalized names are exported.
func sayMyName() {
  print("Hrunkner")
}

func SayHello() { // exported.
  print("Why hello there")
  sayMyName()
}
```

Lua's `requires` versus Go's `import`. Note that `require`'s return values are cached so a file is run at most once, even when require'd many times.

```
local mod = require('mod')  -- Run the file mod.lua.

mod.sayHello()  -- Says hello to Hrunkner.
mod.sayMyName()  -- error: sayMyName only exists in mod.lua:
```

```
import "github.com/wangkuiyi/example/mod"

mod.SayHello()  // Good.
mod.sayMyName()  // Error: sayMyName is not exported.
```

In addition to `require`, Lua has `dofile`, `loadfile`, `loadstring` for dynamically loading. For compiled languages, dynamic loading is hard to implement.

```
-- dofile is like require without caching:
dofile('mod2.lua')  -- Hi!
dofile('mod2.lua')  -- Hi! (runs it again)

-- loadfile loads a lua file but doesn't run it yet.
f = loadfile('mod2.lua')  -- Call f() to run it.

-- loadstring is loadfile for strings.
g = loadstring('print(343)')  -- Returns a function.
g()  -- Prints out 343; nothing printed before now.
```