Lua: Organization

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Friday, February 1, 2019

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

The **Lua** PL originated in 1993 at the Pontifical Catholic University in Rio de Janeiro, Brazil.

Lua's **source tree** is small, easy to include in other projects. It is a popular scripting language for games, LaTeX, Wikipedia, etc.

Characteristics

- Dynamic PL.
- Simple syntax. Very little **punctuation**. Small, versatile feature set.
- **Imperative**.
- Insulated from machine.
- Typing: **dynamic**, **implicit**. **Duck typing**.
- Only eight types: `number`, `string`, `boolean`, `table`, `function`, `nil`, `userdata`, `thread`.
- **First-class functions**.
- Function definitions are executable statements.
- Uses **eager** evaluation (opposite: **lazy** evaluation).

Lua: Fundamentals [1/2]

A **keyword** is a word with special meaning in a PL.

Lua has 21 keywords: `and break do else elseif end false for function if in local nil not or repeat return then true until while`

A word that has the general form of an identifier, but is not legal as an identifier, is a **reserved word**.

In Lua, the reserved words are the same as the keywords. This is true in many other PLs as well—but not in all PLs.

Lua: Fundamentals [2/2]

Miscellaneous Points

- Only values have types; variables are references to values.
- Function `print` does quick & dirty output. `io.write` is preferred.
- "`..`" op: string concatenation, with number-to-string conversion.
- If passing a single table literal or string literal to a function, then you may leave off the parentheses in the function call: `foo "abc"`
- A parameter that is not passed gets the value `nil`.
- Variables inside a function default to global—except parameters & loop counters. Declare local variables using `local`.
- Dot syntax: `t["abc"]` and `t.abc` are the same.
- **Array**: table whose keys are 1, 2, …, *n*.
- Length of array `arr`: `#arr`
- Iterator-based for-in loops (we will write our own iterators):
  - `for k, v in pairs(`*TABLE*`) do` *STMTS* `end`
  - `for k, v in ipairs(`*TABLE_AS_ARRAY*`) do` *STMTS* `end`
- Flow of control not covered now: exceptions, coroutines, threads.

Lua: Organization
Modules — `require` [1/2]

Lua's standard library includes a function `require`.

- Other standard-library functions: `print`, `pairs`, `ipairs`.

Function `require` takes a single string argument. It adds "`.lua`" to the end of this string and treats the result as the filename of a Lua source file. It calls the code in that file, just as if it were a function wrapped in "`function() … end`", like this:

```
function()
   …
   …
end
```

Contents of file

For code from today, see `org.lua` & `mymod.lua`.

The return value of `require` is the return value of the file-as-function—`nil` if nothing is returned.

Lua: Organization
Modules — `require` [2/2]

What is the point?

Suppose file `mymod.lua` contains code that constructs and returns a table. And that table has keys f and `g`, whose associated values are functions. Then we can do this:

```
mymod = require "mymod"

mymod.f()
result = mymod.g(1, 2, "x")
```

We can leave off the parentheses in the call to function `require`.

Now the value of variable `mymod` is the returned table.

The result is a **module**: an external library that can be imported into a program—the kind of thing we would make a header/source pair for in C++. To bring a module into a program, we **require** it.

Lua: Organization
Modules — Writing a Module [1/2]

In the module file, start by creating a local table, which should be empty. You can name this whatever you want, but I think the code is clearest if the name matches the name of the file.

```
local mymod = {}
```

Things to export are table members. Everything else is local.

```
function mymod.add6(n)
  return n+6
end
```

Return the module table at the end of the file.

```
return mymod
```

Modules — Writing a Module [2/2]

Remember that, in a module file, if something is not a module table member and not local, then it is global. That means you are messing with the client code's variables—usually a bad idea.

So if your module requires another module, make the table local:

```
local othermod = require "othermod"
```

Lua: Organization
Modules — Calling a Local Function [1/3]

Sometimes you need to call a local function before its definition. This can be a bit tricky, because keyword `local` actually creates a new local variable when it is executed. Code compiled before the "`local`" will refer to the global of the same name.

```
foo()

local function foo()
  …
end
```

This calls the global version of `foo`.

The local version of `foo` is created here.

Lua: Organization
Modules — Calling a Local Function [2/3]

We can do a "`local`" on a function name before the function's definition. However, a second "`local`" will create a second variable.

```
local foo

foo()

local function foo()
  …
end
```

This calls the local version of `foo` created here.

This creates a second local named "`foo`", which is not the one that is called above.

Lua: Organization
Modules — Calling a Local Function [3/3]

The solution is to do a "`local`" on the function name before the call, then have *no* "`local`" with the function definition.

```
local foo

foo()

function foo()
  …
end
```

This version works!

This redefines the local named "`foo`" that was created above.

If all calls to a local function come after its definition, then you do not need to worry about this issue.

Lua: Organization
Modules — Using a Module

Again, to bring a module in, use the standard function `require`:

```
mymod = require "mymod"
```

You may want to make the module table local:

```
local mymod = require "mymod"
```

Then access module members with the dot operator:

```
io.write(mymod.add6(15).."\n");
```

The code for the module goes in a file whose name is the module name plus the Lua suffix: "`mymod.lua`".

Lua: Organization
Metatables [1/4]

A Lua table can have an associated **metatable**. We use a metatable to implement various special operations involving the original table. For example, Lua uses metatables to do operator overloading. A metatable also allows us to determine what happens when a nonexistent key is accessed.

We will use the latter functionality to simulate the class-object relationship in programming languages like C++.

Let us make a table (which is to become the metatable for another table) and put a function in it.

```
mt = {}  -- Empty table, to be used as metatable

function mt.printYo()
  io.write("Yo!\n")
end
```

Lua: Organization
Metatables [2/4]

Suppose we attempt to get the value corresponding to a key in a table, but that key is not in the table. If the table has a metatable, then, the `__index` item in the metatable is called—so it needs to be a function—with two arguments: the original table and the missing key. The return value of this function call is used in place of the missing value from the original table.

Define `__index` to return the corresponding item in the metatable.

```
function mt.__index(tbl, key)
  return mt[key]
end
```

Lua: Organization
Metatables [3/4]

Think of `mt` as being like a C++ class. Now we create an "object" of that class: another table `t`, whose metatable will be `mt`.

`setmetatable` is another standard-library function.

```
t = {}
setmetatable(t, mt)
```

Table `t` has no member `printYo`.

So calling `t.printYo()` invokes `mt.__index(t, "printYo")`, which returns `mt.printYo`, which is then called.

```
t.printYo()  -- Print "Yo!"
```

Lua: Organization
Metatables [4/4]

We can simulate a constructor by adding a creation function to the metatable. We might call such a function "`new`".

```
function mt.new()
  local obj = {}
  setmetatable(obj, mt)
  return obj
end
```

Then we can create our "object" by calling the above function.

```
t = mt.new()  -- Create new object, using mt as "class"
t.printYo()   -- Print "Yo!"
```

Lua: Organization
Colon Operator [1/3]

PLs like C++ make a strong distinction between ordinary functions and member functions: a member function knows which object it is a member of; it accesses the object via the "`this`" pointer.

In Lua, a function that lies in a table is not special. It is simply an ordinary function that happens to be a value associated with some table key. The function *does not know it is in a table*, and it certainly cannot tell which table it lies in.

It is often useful for a function to have this information, since then it can access other values in the table. We can give a function this information by passing the table as a parameter.

```
-- Add the given value to the x member of the table.
function t.set_x(self, val)
  self.x = val
end
```

Lua: Organization
Colon Operator [2/3]

```
-- Add the given value to the x member of the table.
function t.set_x(self, val)
  self.x = val
end
```

We can call function `set_x` as follows.

```
t.set_x(t, 17)  -- Set t.x to 17
```

But the above is redundant: `t` is specified twice. So Lua offers a shorthand, using the **colon operator**.

```
t:set_x(17)     -- Increase t.x by 17
```

Lua: Organization
Colon Operator [3/3]

The colon operator is particularly useful when a function is in a metatable. Suppose table `t` has metatable `mt` as before.

```
function mt.print_x(self)
  io.write(self.x.."\n")
end
```

Now we can print `t.x` as follows.

```
t:print_x()
```

Using metatables and the colon operator, we can do object-oriented programming in Lua.

However, Lua offers other functionality that we may want to use instead: *closures*.

Lua: Organization
Closures [1/3]

A **closure** is a function that carries with it (some portion of) the environment in which it was defined. Closures offer a simple way to do some of the things we might do with an object in traditional C++/Java OO style.

Here is a function that returns a closure.

```
-- make_multiplier
-- Return function that multiplies by the given k.
function make_multiplier(k)
  local function mult(x)
    return k*x
  end
  return mult
end
```

Lua: Organization
Closures [2/3]

Function `mult` is an ordinary function that returns its parameter multiplied by `k`. But what is `k`? It is the parameter of `make_multiplier`. The return value of `make_multiplier` is a closure, since it contains a copy of the `k` that was passed into this particular call to `make_multiplier`.

If we call `make_multiplier` several times, we can get closures with different values of `k`.

```
times2 = make_multiplier(2)  -- Times-2 function
triple = make_multiplier(3)  -- Times-3 function

io.write(times2(7).."\n");   -- Prints "14"
io.write(triple(10).."\n");  -- Prints "30"
```

Lua: Organization
Closures [3/3]

Using traditional OO style, we could do something like this by creating objects with a data member `k`. We could set the value of `k` in a constructor and use it in a member function `mult`.

But closures are a bit simpler. So the existence of closures means we have less need for objects. In particular, if an object exists only to support one member function, which accesses various data members of the object, then we might be better off using a closure instead of an object.

Closures are found in a number of PLs. Since the 2011 standard, C++ has had closures, in the form of *lambda functions*.

See `closure.cpp`.