LabLua
programming language research

Programming in Lua – Getting Started

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

Getting Lua

- The best way to get Lua for a beginner is through your operating system's package manager
- Make sure you are installing Lua 5.2, this is what we will be using in this course
- We will edit Lua files in a regular text editor, and run them through the console using the Lua interpreter
- A comprehensive list of editors (and full-fledged IDEs with Lua support) is at http://lua-users.org/wiki/LuaEditorSupport

The Lua REPL

- If we run the Lua interpreter without a source file it puts us in the Read-Eval-Print Loop, or the **REPL**
- Our first interactions will be through the REPL; it lets us input Lua statements, and evaluates them for us
- The REPL is good for experimentation, but it forgets anything we defined in it after it exits
- We do not want to lose everything, so we will put anything we do not want to lose in a separate **definitions** script, and run it inside the REPL

Chunks and statements

- At the basic level, a Lua program is composed of *statements*: definitions, assignments, conditionals, loops, function calls...
- A sequence of statements forms a *chunk*; they are the unit of variable scoping in Lua, so are important when dealing with local variables and function parameters
- The body of a Lua file is a chunk; the body of a *for* or *while* loop is a chunk; the body of a function definition is a chunk
- Everything the REPL evaluates is also its own chunk

Separating statements

- The syntax of Lua is simple enough that it does not need separators to know when a statement ends and another begins, even if they are on the same line:

```
> a = 1 b = 2 print(a, b)
1       2
```

- But it is good form to use a semicolon if you wish to put several statements in the same line, as a courtesy to the programmer that has to read the code:

```
> a = 1; b = 2; print(a, b)
1       2
```

Running a script in the REPL

- We can define functions in the REPL easily enough, and it will even let us do the definition in multiple lines, but it is much better to put them in another file

- Let us create a “defs.lua” file and put this definition of a factorial function in it:

```
function fact(n)
    if n < 2 then
        return 1
    else
        return n * fact(n - 1)
    end
end
```

- We can load this definition in our REPL and call fact:

```
> dofile("defs.lua")
> print(fact(5))
120
```

Basic functions

- Both `print` and `dofile` are built-in functions
- The Lua syntax for calling functions should be familiar to you, but there are some twists that we will see shortly
- Most built-in functions live in different namespaces; the `math` namespace has several mathematical functions, such as `math.sin` and `math.sqrt`

```
> print(math.sin(math.pi / 3))
0.86602540378444
> print(math.sqrt(3) / 2)
0.86602540378444
>
```

Global variables

- Lua has variables, like any imperative programming language, and they are global by default

- You do not need to declare global variables; just assign to them, and use them; their “scope” is any chunk that is evaluated, unless *shadowed* by a local variable of the same name (more on that later)

```
> print(x)
nil
> x = 2
> print(x)
2
```

- The default value of a global variable is the special value *nil*

A lexical interlude, part 1

- Lua variable names can be any sequence of letters (ASCII), digits, and underscores that does not begin with a digit

- In particular, _ (a single underscore) is a valid variable name, and useful as a “dummy” variable that you assign to but not use

- Avoid variable names starting with _ followed by uppercase letters, these are reserved for Lua (_VERSION, _ENV, _G, etc.)

- Reserved words cannot be used as variable names:

```
and       break     do      else    elseif    end     false
for       function  if      in      local     nil     not
repeat    return    then    true    until     while   or
goto
```

A lexical interlude, part 2

- Single-line comments (like C++'s //) start with `--` (two hyphens), and run until the end of the line

- Block comments (like C's /* */) start with `--[[` and end with `]]`

- As a corner case, `---[[` starts a single-line comment and not a block comment; this is useful for "turning off" block comments

```
--[[
print(x)     -- this line is commented out
--]]

---[[
print(x)     -- this line is not
--]]
```

Quiz

- Which of the following names are valid variable names? Why/why not?

```
--- _end End end until? Nil NULL
```