Lua: Fundamentals
Lua: Modules

CS 331 Programming Languages

Lecture Slides

Wednesday, January 29, 2025

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

Unit Overview
The Lua Programming Language

Topics

- ✓ Introduction to survey of programming languages
- ✓ PL feature: compilation & interpretation
- ✓ PL category: dynamic PLs
- ✓ Introduction to Lua
  - Lua: fundamentals
  - Lua: modules
  - Lua: objects
  - Lua: advanced flow

Review

Review
PL Category: Dynamic PLs

Script: program associated with a software package, used to extend functionality, automate operations, or customize. Written in the package's **scripting language**.

Out of early shell scripting languages and associated text-processing tools grew the full-featured PL **Perl** (1987).

Similar PLs: **Python** (1991), **Lua** (1993), **Ruby**, **JavaScript**, and **PHP** (1995). These are **dynamic programming languages**.

Typical Characteristics

- Dynamic typing (types determined & checked at runtime).
- Just about everything is modifiable at runtime.
- Little text overhead in code.
- High-level.
- A batteries-included approach.
- **Imperative** and block-structured, with support for OOP.
- Mostly interpreted, with compilation to byte code as an initial step.

Introduction to Lua — History, Characteristics

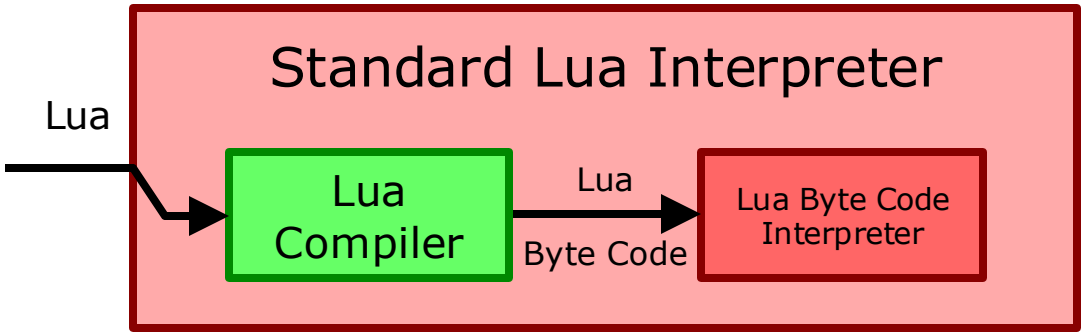
The **Lua** PL originated in 1993 at the Pontifical Catholic University in Rio de Janeiro, Brazil.

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. Little **punctuation**. Small, versatile feature set.
- **Imperative**.
- Insulated from machine.
- Typing: **dynamic**, **implicit**. **Duck typing**.
- Exactly eight types: `number`, `string`, `boolean`, `table`, `function`, `nil`, `userdata`, `thread`.
- **First-class functions**.
- Function definitions are executable statements.
- Does **eager evaluation** (alternative: **lazy evaluation**).

Lua is nearly always interpreted. The interpreter in the standard Lua distribution compiles Lua to **Lua byte code**, which is system-independent. This byte code is then interpreted directly by the runtime system.



Standard filename suffix for Lua source files: "`.lua`".

Standard Lua interpreter includes an **interactive environment**.

Lua: Fundamentals

Lua: Fundamentals
Overview

Now we begin a systematic look at the Lua programming language.

We start with Lua's lexical structure. Then we look at the following topics. To avoid rehashing the tutorial, these will be presented as summaries.

- Values & Expressions
- Output
- Functions
- Tables
- Arrays
- Flow of control

The material for this topic is also covered in a Lua source file with lots of comments. Find this in the Git repo.

See `fund.lua`.

Lua: Fundamentals
Lexical Structure — Comments

Like many other PLs, Lua has both single-line comments and multiline comments.

Single-line comments: "--" to end-of-line:

```
-- This is a comment
```

Multiline comments: "--[" + zero or more equals-signs (=) + "[". End with "]" + same number of equals-signs + "]".

```
--[==[This is a
comment]==] this_is_not_a_comment = 3
```

Lua: Fundamentals
Lexical Structure — Keywords & Identifiers

Lua is **case-sensitive**: `abc`, `Abc`, and `ABC` are different. Case of letters has no special meaning in Lua.

The boldfaced terms on this slide are not specific to Lua.

Lua has 21–22 **keywords** (words with special meaning):

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

*The `goto` keyword was added in Lua 5.2. LuaJIT is still based on Lua 5.1.

A Lua **identifier** (name) is as in "C" : contains underscores (_), letters, digits, does not begin with a digit, and is not a keyword.

A word that has the general form of an identifier, but is not allowed as an identifier, is a **reserved word**. So the Lua reserved words are the keywords. This is common, but not universal.

Lexical Structure — String Literals

A **literal** is a representation of a fixed value in source code. The value itself is represented, not an identifier bound to the value, and not a computation whose result is the value.

This term is not specific to Lua.

Some C++ Literals

Literal	Type
`42`	`int`
`42.5`	`double`
`42.5f`	`float`
`true`	`bool`
`'A'`	`char`
`"goat"`	`char[]`

Two kinds of string literals in Lua.

Quoted strings use single or double quotes.

```
aa = "hi"
ba = 'ho"\n'  -- Backslash escape: \n = newline
```

Multiline strings use brackets & equals, like multiline comments.

```
cc = [===[Hello
there! Here is a quote mark inside a string: "]===]
```

Separating lexemes with whitespace is allowed, but not required …

```
xyz=12      -- Same as xyz = 12
```

… except where it clearly matters.

```
do return   -- "do" keyword followed by "return" keyword
doreturn    -- Identifier
```

Newlines and indentation are usually not syntactically significant.

Newlines are treated the same as other whitespace, except:

- A newline ends a single-line comment.
- A newline is illegal in a quoted string (but `"\n"` is fine).
- A newline represents itself in a multiline string—except when the first character is a newline; such a newline is skipped.

Summary

- Only values have types; variables are references to values.
- Function `type` returns a string giving the type of its argument.
  - For example, `type(2+3)` returns the string `"number"`.
- Type errors are flagged at runtime, when the statement containing the error is executed. An exception is raised.
- Arithmetic expressions & comparisons are mostly as usual.
  - Inequality operator: "`~=`".
- **Multiple assignment**: `a, b, c = 3, 4+d, 5`
- Booleans: `true`, `false`, `and`, `or`, `not`.
- Function `tonumber` converts its argument to a number.
- Function `tostring` converts its argument to a string.
- "`..`" operator does string concatenation, with implicit number-to-string conversion.

Summary

- `io.write` does output, with its argument converted to a `string` where possible; no newline is added. `io.write` normally outputs to the standard output, but it an also be used to output to a file.
- `print` also does output, with arguments separated by tabs and a newline added at the end. `print` only sends to the standard output.
- My convention: use `io.write` for normal application output. Use `print` for quick & dirty output—like debugging printout.

Summary

- Main program is code at global scope.
- A function definition begins with the keyword `function`. This is an executable statement.
- Call functions as usual.
  - If passing only a single string literal or table literal, then the parentheses may be left off: `foo("abc")` → `foo "abc"`
- A parameter that is not passed gets the value `nil`.
- Return values from functions as usual.
  - The value of a function call that does not return anything is `nil`.
  - Multiple values can be returned: `return x, 42, y+3`
    Capture these with multiple assignment: `a, b, c = ff()`
- **First-class functions**.
- Omit the function name to create an unnamed function, referred to as a **lambda function**:

```
gg(function(n) return n*n end)
```

Summary

- Maps/dictionaries, arrays, objects, and classes are implemented using a single Lua feature: **table**, a key-value structure implemented internally as a hash table.
- Table literals use braces, entries separates by commas. Key-value pair is key in brackets, equals sign, value: `{ …, ["abc"]=56, … }`
- Access values using brackets, as in C++/Java: `t["abc"] = 4`
- If a key looks like an identifier, then we can use dot syntax: `t["abc"]` and `t.abc` are the same.
- Delete a key from a table by setting the associated value to `nil`: `t["abc"] = nil`
- We can mix types of keys, values.
- We can put functions in tables.
  - We can declare a function as a table member: `function t.foo(x) …`
- Loop over all key-value pairs in a table with `pairs` (example soon).
- Tables are also used to implement operator overloading (not covered here).

Summary

- When a Lua table's keys are consecutive positive integers starting at one (not zero!), we call it an **array**. This usage is only for Lua!
- Array literal: list values in braces without keys. Indices start at one.
  `arr = { 7, "abc", fibo, 5.34 }`
- Length of array `arr`: `#arr`
- Loop over array items, in order, with `ipairs` (example soon).

Summary

- `if` *COND* `then` *STMTS* `end`
  - `false` & `nil` are **falsy** (treated as `false`). All other values are **truthy**.
- `if` *COND* `then` *STMTS* `else` *STMTS* `end`
- `if` *COND* `then` *STMTS* `elseif` *COND* `then` *STMTS* … `end`
  - "`else if`" is legal, but requires an extra "`end`", which "`elseif`" avoids.
- `while` *COND* `do` *STMTS* `end`
- `repeat` *STMTS* `until` *COND*: like C `do` … `while`, condition flipped.
- `for` *VAR*`=`*FIRST*`,` *LAST* `do` *STMTS* `end`
- `for` *VAR*`=`*FIRST*`,` *LAST*`,` *STEP* `do` *STMTS* `end`
- `break`: as in C. (There is no "continue".)
- Iterator-based for-in loop. Examples:
  - `for k, v in pairs(`*TABLE*`) do` *STMTS* `end`
  - `for k, v in ipairs(`*TABLE_AS_ARRAY*`) do` *STMTS* `end`

  We will eventually write our own iterators.
- Other (not covered yet): coroutines, exceptions.

Lua: Modules

Lua: Modules
Introduction

A **module** is an importable library that is **encapsulated**—handled as a single entity by a programming language.

For code from this topic, see `mod.lua` & `mymodule.lua`.

Most modern PLs support modules. A module is usually stored in a separate file and **imported** (brought into a program) with a command like "import" or "require".

(C does not support modules. Modules were added to C++ in the the 2020 Standard.)

Lua's standard library is loaded automatically; no import is needed. But we can write our own modules, and we must import these.

Before we talk about writing modules, we cover two more pieces of Lua: keyword `local` and standard-library function `require`.

Lua: Modules
Two Pieces — local [1/6]

A **local** variable is accessible only inside a particular function or other construct. **Global** variables do not have this restriction.

Lua variables default to global, except for function parameters and loop counters. To create a new variable that is local to a function, put keyword `local` before its first use. The variable is then local in future uses in that function.

```
function ff()
    k = 3          -- Set global variable k to 3
    local n = 5    -- Create local variable n; set it to 5
    n = n+1        -- Local n; value is now 6
    …
end
-- The above local variable n cannot be accessed here
```

Lua: Modules
Two Pieces — local [2/6]

Multiple variables can be declared local in a single statement. Setting values in a `local` statement is optional.

```
local n, qq, www
n = 5          -- Set local variable n to 5
local a, b, c = 2, 4, 10
```

Each `local` creates a *new variable*, accessible after the `local`; so avoid using `local` more than once with the same identifier.

```
x = 7          -- Sets global variable x
local x = 3    -- Creates & sets local variable x
local x        -- Creates another local variable named x
-- Here, the value of x is nil
```

Lua: Modules
Two Pieces — local [3/6]

Use `local` before a function definition to create a local function.

```
function gg()
    local function hh(x)
        …
    end
    hh(3)
end
-- The above local function hh cannot be called here
```

Two Pieces — local [4/6]

Problem. How do we call a local function before its definition?

```
local function mm()
    foo()
end

local function foo()
    …
```

This calls the global `foo`.

The local `foo` is created here.

Of course, we could just define function `foo` first. But suppose we have mutually recursive local functions (each of them calls the other). Then this issue is unavoidable.

Two Pieces — local [5/6]

Problem. How do we call a local function before its definition?

```
local foo

local function mm()
    foo()
end

local function foo()
    …
```

This calls the local `foo` declared above.

A new local `foo` is created here. It is different from the one called above.

Two Pieces — local [6/6]

Problem. How do we call a local function before its definition?

Solution. Do a `local` on the function name before the call, then have *no* `local` with the function definition.

```
local mm, foo

function mm()
    foo()
end

function foo()
    …
```

This redefines the local `foo` that was created and called above—without creating another `foo`.

Lua: Modules
Two Pieces — require [1/2]

Our second piece is `require`, a function in Lua's standard library. It takes a single string argument. It appends "`.lua`" to 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 that is `required`

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

The file will be read only once. Subsequent calls to `require` with the same filename simply return the return value of the first call.

Lua: Modules
Two Pieces — require [2/2]

Since `require` always takes a single string argument, we can leave off the parentheses in the function call.

```
abc = require "xyz"  -- Calls file xyz.lua as a function
                     -- Sets abc to its return value
```

In our file-as-function, we can declare variables and functions to be local, just as in an ordinary function.

```
local a = 23
local b = 34
return a+b
```

Contents of file that is `required`

Lua: Modules
Writing a Module [1/2]

Now we discuss the conventional way to use `local` and `require` to create and make use of a Lua module.

Store a module in a file named after the module. For example, module *quark* would be in `quark.lua`.

Begin the module file by creating an empty local table. It is not a bad idea to name this table after the module, too.

```
-- File quark.lua
-- Source for module quark
local quark = {}  -- Table that will contain module
                  --  members
```

Lua: Modules
Writing a Module [2/2]

Not a Lua-specific term. To **export** something is to make it available elsewhere.

Things to *export* are module members. *Everything else* is local.

We can make a function a module member in its definition.

```
local function ff(n)          -- ff is not exported
    …
end

function quark.gg(a, b, c)    -- gg is exported
    ff(a+b+c)
end
```

End the module file by returning the table.

```
return quark
```

Lua: Modules
Using a Module [1/2]

To import a module, use `require`. Save the return value. It is not a bad idea to save the module table in a variable named after the module.

```
quark = require "quark"
```

Access module members using the dot syntax for table members.

```
quark.gg(2, 4, 10)
```

When requiring a module inside another module, make the variable holding the module table local. ("*Everything else* is local.")

```
local quark = require "quark"
```

Lua: Modules
Using a Module [2/2]

If you want to use a module member without having to add *MODULENAME*. before it, then you can set a variable equal to the module member.

```
quark.gg(1, 2, 3)  -- This works, but what if I think
quark.gg(4, 5, 6)  --  it is too verbose?

gg = quark.gg      -- Use a variable with a nicer name
gg(1, 2, 3)
gg(4, 5, 6)
```

This works because functions are first-class.

The above is much like doing "`using quark::gg;`" in C++.

Lua: Modules
CODE

TO DO

- Write a Lua module and a program that uses it.

Done. See `mymodule.lua` *&* `mod.lua`*.*