An Introduction to the Lua Programming Language

Davis Claiborne

NCSU LUG

February 10, 2018

Linux Users Group
at NC State University

What is Lua?

Open source scripting language developed in Brazil

Primarily known for

- Speed (for an interpreted language)
- Simplicity
- Embedability
- Portability

Where is Lua used?

Lua can be found embedded in many different areas:

- Web
  - **MediaWiki** templates [1]
  - Internet servers such as **Apache** [2] and **NGINX** [3]
  - **Moonshine** is a Lua VM for browsers [4]
- Software
  - **VLC** for custom scripting [5]
  - **LuaTeX** is an extended version of TeX [6]
  - Network diagnostic tools, including **Nmap** [7] and **Wireshark** [8]
  - **Torch** machine learning uses Lua [9]
- Games
  - Many games, such as **World of Warcraft** [10], **Roblox** [11], and more all allow creating plugins using Lua
  - Number 1 most used langauge in game dev [12]
- And many more... [13]

How Fast is Lua?

Lua is one of the fastest interpreted languages around [14]

A few notes on this test:

- It only uses one test application, so it's not an ideal showcase
- Test is comparing embeded implementations of languages

Lua can be made even faster with LuaJIT [15]

- LuaJIT is *at least* two times faster, can be >64x for some tests
- Exposes FFI for even greater performance increases

What Does Lua's Syntax Look Like?

Lua's syntax is pretty simple and very similar to JavaScript. This is not an all-inclusive list; just a quick run-down.

```
-- Two dashes represent single-line comments
-- Lua is dynamically-typed and duck-typed, so declaring
-- a variable involves no types
languageName = 'lua'
avagadrosNumber = 2.2e23
boolean = true

--[[
Blocks comments are done with two square brackets, with
an optional number of `=' in between, allowing for
nesting of block comments.
]]
```

```lua
--[=[ There are 5 main types in Lua:
        * boolean
        * number
        * string
        * function
        * table
(Lua actually has 8 types; I'm ignoring the rest for now)
]=]

-- Using and declaring functions is simple
function foo( x )
        print( x )
end

foo( "test" ) -- Outputs "test" to stdout
```

How Embeddable is Lua?

Lua can be used on microcontrollers with eLua [16]

Lua is very easy to embed in other languages, including: [12]

- C
- C++
- Java
- Fortran
- Ada
- ...

Lua is a good choice for many applications due to its small size, speed, small memory footprint, etc. [17]

It is possible to embed Lua without the compiler to save memory [18]

How Portable is Lua?

Lua is written entirely in ANSI C [19]

High emphasis on being low-profile:

From *Programming in Lua*: [20]

"Unlike several other scripting languages, Lua does not use POSIX regular expressions (regexp) for pattern matching. The main reason for this is size: A typical implementation of POSIX regexp takes more than 4,000 lines of code. **This is bigger than all Lua standard libraries together.**"

From Luiz Henrique de Figueiredo, Lua Team member: [16]

"Very early on in the development of Lua we started using the question '**But will it work in a microwave oven?**' as a half-serious test for including features while avoiding bloat."

The entire size of the Lua interpreter and base libraries can fit in well under 1 MB [18]

Notable Aspects of Lua: Coroutines

Coroutines allow for intuitive async code

```
-- Non-async code
function foo()
        print( "first" )

        -- How to suspend execution until later?
        print( "third" )
end

function bar()
        print( "second" )
        print( "fourth" )
end

foo() -- "first", "third"
bar() -- "second", "fourth"
```

Is there any way to get these functions to pause and resume easily?

Coroutines create separate threads for each function, allowing for easy and intuitive async events

```lua
function foo()
	print( "first" )
	coroutine.yield() -- Suspends thread until resumed
	print( "third" )
end

function bar()
	print( "second" )
	coroutine.yield()
	print( "fourth" )
end

co1 = coroutine.create( foo )
co2 = coroutine.create( bar )

coroutine.resume( co1 ) -- "first"
coroutine.resume( co2 ) -- "second"
coroutine.resume( co1 ) -- "third"
coroutine.resume( co2 ) -- "fourth"
```

Notable Aspects of Lua: Global by Default

Lua features variables that are global by default[1], and block-local

```
function foo()
        local bar = 'this is local'
        baz = 'this is global'

        print( bar ) -- "this is local"
        print( baz ) -- "this is global"
end

foo()
print( bar ) -- "nil"
print( baz ) -- "this is global"
```

Undefined variables do not cause errors; instead they return "nil"[2]
Local values are preferable for performance and complexity reasons

[1] This can be protected against; implementation will follow here
[2] This is considered by most to be one of the major flaws of Lua

Notable Aspects of Lua: Tables

Tables are the only memory container format in Lua

```
my_table = {
        string = 'asdf', -- Named keys
        1, -- Non-named keys are automatically integers
        3,
        5,
}

print( my_table.string ) -- "asdf"
print( my_table['string'] ) -- also "asdf" (both ways work)
print( my_table[1] ) -- "1" (Note: tables start at 1 in Lua)
print( my_table[2] ) -- "3"
print( my_table[3] ) -- "5"
print( my_table.1 ) -- Syntax error; not a string key
```

Notable Aspects of Lua: Tables

Almost anything in Lua can act as a table key, even other tables

```lua
function foo()
end

other_table = {
        [foo] = "function foo",
        ["1"] = "string 1", -- Different than numeric 1
        foo, -- Integer that references a function
        [true] = "true value",
        [my_table] = "my_table is the key",
}

print( other_table.foo )       -- "nil"
print( other_table[foo] )      -- "function foo"
print( other_table['1'] )      -- "string 1"
print( other_table[1] )        -- "function: 0x......."
print( other_table[true] )     -- "true value"
print( other_table[my_table] ) -- "my_table is the key"
```

Notable Aspects of Lua: Tables

Tables can even be cyclic

```
cyclic1 = {}
cyclic2 = {}

cyclic1[1] = cyclic2
cyclic2[1] = cyclic1

print( cyclic1, cyclic2 )             -- table: a, table: b
print( cyclic2[1], cyclic1[1] )       -- table: a, table: b
print( cyclic1[1][1], cyclic2[1][1] ) -- table: a, table: b
```

Notable Aspects of Lua: Global Variable Table

All global variables are stored in a special table, "_G"

```
globalVariable = 'adsf'
print( _G.globalVariable ) -- 'asdf'
```

This table contains not only all global variables, but also all base-library functions, such as `print`.

Using a table to store global variables allows for powerful customizability through **metamethods**

Notable Aspects of Lua: Metamethods

Metamethods are special functions, in tables called **metatables**, that allow customization of tables

These allow for OOP-like behavior and more

Metamethods exist for:

- Addition
- Subtraction
- Multiplication
- Concatenation
- and more...

Metamethods can also be used for sandboxing

Live demo

```lua
point = {}

function point.new( x, y )
        return setmetatable( { x = x or 0, y = y or 0 }, point )
end

-- Invoked when addition occurs
function point.__add( a, b )
        return point.new( a.x + b.x, a.y + b.y )
end

-- Invoked when table is called like a function
function point:__call( x, y )
        return point.new( x, y )
end

-- Applies metamethods; nothing special about table before this
-- A table can have any table as its metatable, even itself
setmetatable( point, point )
```

```lua
pointA = point()
pointB = point( 3, 3 )

print( pointA ) -- table: 0x......

-- Invoked when table is concatted
function point:__tostring()
        -- Note implicit self (: vs . in function name)
        -- Is the same as point.__tostring( self, ... )
        return "( " .. tonumber( self.x ) .. ", "
                    .. tonumber( self.y ) .. " )"
end

print( pointA ) -- "( 0, 0 )"
print( point.__tostring( pointA ) ) -- "( 0, 0 )"
print( pointB ) -- "( 3, 3 )"

pointC = pointA + pointB
print( pointC ) -- "( 3, 3 )"

pointD = point( -3, 4 )
pointE = pointC + pointD
print( pointE ) -- "( 0, 7 )"
```

Using Metamethods to Prevent Accidental Globals

```lua
declaredGlobals = {}

function declare( name )
        declaredGlobals[name] = true
end

setmetatable( _G, {
        -- Called every time a new key is added to a table
        __newindex = function( tab, key, value )
                assert( declaredGlobals[key],
                        "Error: value not declared"
                )
                -- Directly set the value
                -- (assigning would cause infinite loop)
                rawset( tab, key, value )
        end,
} )

foo = 3 -- Error: value not declared...
declare( 'foo' )
foo = 3
```

Notable Aspects of Lua: Proper Tail calls

Proper tail calls are good for recursive algorithms

Stack-overflow **cannot** occur due to a proper tail call

A tail call is defined as "when a function [only] calls another [function] as its last action." [21]

Live demo

```lua
-- Improper tail call, as it's multiplying; not "just" tail call
function factorial( n )
        if n == 0 then
                return 1
        else
                return n * fact( n - 1 )
        end
end

factorial( -1 ) -- Stack overflow

-- Proper tail call implementation of factorial
function factorial( n, prod )
        prod = prod or 1

        if n == 0 then
                return prod
        else
                return factorial( n - 1, n * prod )
        end
end

factorial( -1 ) -- Infinite loop
```

Notable Aspects of Lua: First-class functions

Functions are **first-class** values

This basically means that functions can be used as arguments, return values, etc. Essentially, functions can be treated just like any variable.

Consider the following example:

```
family = { "mom", "father", "sister", "son" }

-- Note the "anonymous" function as a parameter
table.sort( family, function( string1, string2 )
        return #string1 < #string2 -- # means "the length of"
end )

for i = 1, #family do
        print( family[i] )
end
-- "mom", "son", "sister", "father"
```

Notable Aspects of Lua: Closures and Lexical Scoping

A **closure** is a type of function with full access to its calling environment. This environment is called its **lexical scope**.

```
function sortNames( names )
        table.sort( names, function( string1, string2 )
                return #string1 < #string2 -- (# is "length of")
        end )
end

family = { "mom", "father", "sister", "son" }
sortNames( family )

for i = 1, #family do
        print( family[i] )
end
-- "mom", "son", "father", "sister"
```

What type of value is `names` inside of the anonymous sorting function? Is it local or global?

Notable Aspects of Lua: Upvalues

Trick question! It's an "external local variable" or "**upvalue**"[22]

Upvalues can be used with great effect, along with functions, to produce unique behaviors

```
function newCounter()
        local i = 0
        return function()
                i = i + 1
                return i
        end
end

c1 = newCounter()
print( c1() ) -- "1"
print( c1() ) -- "2"

c2 = newCounter()
print( c2() ) -- "1"
print( c1() ) -- "3"
```

Implementations and Tools for Lua

Standard Lua: Currently in version 5.3 [12]; first widespread use of register-based virtual machine [18]

LuaJIT: JIT-based implementation; hybrid of Lua 5.1 and 5.2 [15]

LuaRocks: The most popular package manager for Lua [24]

Moonscript: More symbolic language that compiles to Lua [25]

LuaCheck: Code linter; can check for accidental globals [26]

SciLua: Collection of libraries intended for researchers [27]

Julia: Language with nearly identical Lua syntax; inteded for scientific use. Features parallel execuetion, arbitrary accuarcy, and more [28]

LuaRocks

LuaRocks is the most popular package manager for Lua.

Includes pure-Lua libraries as well as C bindings.

Contains over 2K modules

Moonscript

Moonscript features a number of differences from Lua [25]

Differences:

- Variables local by default
- Significant whitespace
- Built-in OOP

Moonscript:

```
-- Moonscript features implicit returns
sum = (x, y) -> print "sum", x + y
```

Equivalent Lua:

```
function sum( x, y )
        print( "sum" )
        return x + y
end
```

This code in Moonscript

```
-- Moonscript
evens = [i for i=1,100 when i % 2 == 0]
```

Gets compiled to this

```
-- Lua
local evens
do
  local _accum_0 = { }
  local _len_0 = 1
  for i = 1, 100 do
    if i % 2 == 0 then
      _accum_0[_len_0] = i
      _len_0 = _len_0 + 1
    end
  end
  evens = _accum_0
end
```

SciLua

Seeks to bridge the gap between the use of high-performance languages and scripting languages in the scientific community

Combines several libraries for scientific and statistical use

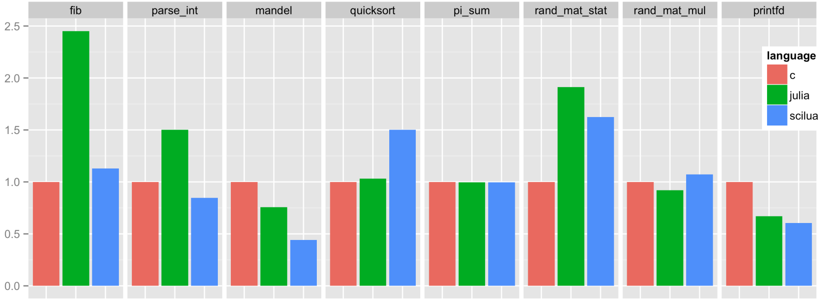

Figure: Relative speed comparison of C, Julia, and SciLua (LuaJIT) [27]

Julia

Julia new language primarily for machine learning

Combines aspects of Python, Lua, and C and Fortran

Example: [29]

```
# "Map" function.
# Takes a string. Returns a Dict with the number of times each
# word appears in that string.
function wordcount(t)
    words=split(t,[' ','\n','\t','-','.',',',':',';'];keep=false)
    counts=Dict()
    for w = words
        counts[w]=get(counts,w,0)+1
    end
    return counts
end
```

References I

[1] Why "Lua" is on everybody's lips, and when to expect MediaWiki 1.19 https://en.wikipedia.org/wiki/Wikipedia:Wikipedia_Signpost/2012-01-30/Technology_report

[2] Apache HTTP Server https://httpd.apache.org/docs/trunk/mod/mod_lua.html

[3] NGINX (OpenResty) https://openresty.org/

[4] Moonshine http://www.moonshinejs.org/

[5] Extending VLC with Lua http://www.coderholic.com/extending-vlc-with-lua/

[6] LuaTeX http://luatex.org/

[7] Nmap https://nmap.org/book/nse-language.html

[8] Wireshark https://wiki.wireshark.org/Lua

[9] Torch http://torch.ch/

[10] World of Warcraft Scripting http://www.wowwiki.com/Lua

[11] Roblox Scripting http://wiki.roblox.com/?title=Scripting

[12] Why choose Lua? https://www.lua.org/about.html#why

[13] Where is Lua Used https://sites.google.com/site/marbux/home/where-lua-is-used

[14] Game Scripting Languages Benchmarked https://github.com/r-lyeh-archived/scriptorium

[15] LuaJIT https://luajit.org/

References II

[16] eLua http://www.eluaproject.net/

[17] Lua on Cell Phones http://lua-users.org/lists/lua-l/2007-11/msg00248.html

[18] The Implementation of Lua 5.0 https://www.lua.org/doc/jucs05.pdf

[19] Lua 5.1 Reference Manual https://www.lua.org/manual/5.1/manual.html

[20] Why Lua doesn't implement POSIX RegEx https://www.lua.org/pil/20.1.html

[21] Proper Tail Calls https://www.lua.org/pil/6.3.html

[22] Upvalues https://www.lua.org/pil/6.1.html

[23] The Implementation of Lua 5.0 https://www.lua.org/doc/jucs05.pdf

[24] LuaRocks https://luarocks.org/

[25] Moonscript https://moonscript.org/

[26] LuaCheck https://github.com/mpeterv/luacheck

[27] SciLua http://scilua.org/

[28] Julia https://julialang.org/

[29] Wordcount Julia Example https://github.com/JuliaLang/julia/blob/master/examples/wordcount.jl

The End