



PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO



Scripting with Lua

Roberto Ierusalimschy

Curry On 2017

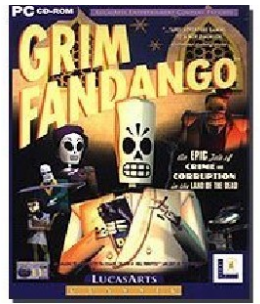
Scripting

- Scripting language x dynamic language
 - scripting emphasizes inter-language communication
- Program written in two languages.
 - a scripting language and a system language
- System language implements the hard parts of the application.
 - algorithms, data structures
 - little change
- Scripting *glues* together the hard parts.
 - flexible, easy to change

Scripting

- Scripting protects the application from the programmer.
 - or vice versa
- Protects the hardware in embedded systems.
- Protects integrity in sensitive software.
 - e.g., World of Warcraft: *“Addons cannot do anything that Blizzard doesn't want them to.”*

Scripting in Grim Fandango



“[The engine] doesn't know anything about adventure games, or talking, or puzzles, or anything else that makes Grim Fandango the game it is. It just knows how to render a set from data that it's loaded and draw characters in that set. [...]

“The real heroes in the development of Grim Fandango were the scripters. They wrote everything from how to respond to the controls to dialogs to camera scripts to door scripts to the in-game menus and options screens. [...]

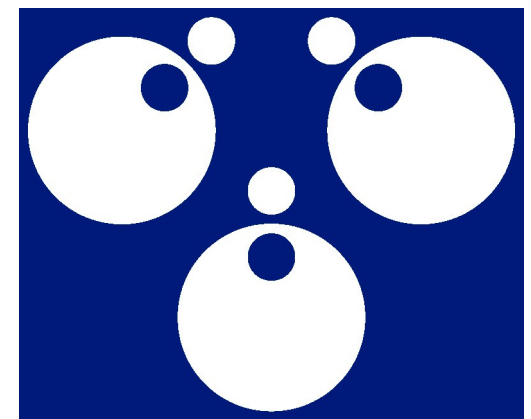
“A TREMENDOUS amount of this game is written in Lua. The engine, including the Lua interpreter, is really just a small part of the finished product.”

Bret Mogilefsky

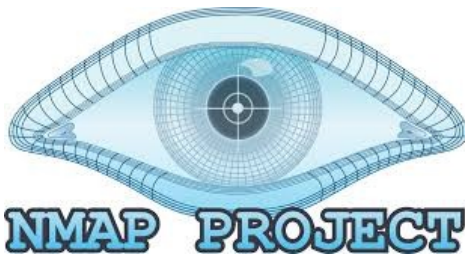
Assistant Designer and Lead Programmer of
Grim Fandango

Scripting (with Lua) in the Wild

- Widely used in several niches
 - Scripting for applications
 - Embedded systems
 - Games



LuaTeX



```
if not _params.STD then
  assert(loadstring(config.get("LUA.LIBS.STD"))())
  if not _params.table_ext then
    assert(loadstring(config.get("LUA.LIBS.table_ext"))())
    if not __LIB_FLAME_PROPS_LOADED__ then
      __LIB_FLAME_PROPS_LOADED__ = true
      flame_props = {}
      flame_props.FLAME_ID_CONFIG_KEY = "MANAGER.FLAME_ID"
      flame_props.FLAME_TIME_CONFIG_KEY = "TIMER.NUM_OF_SECS"
      flame_props.FLAME_LOG_PERCENTAGE = "LEAK.LOG_PERCENTAGE"
      flame_props.FLAME_VERSION_CONFIG_KEY = "MANAGER.FLAME_VERSION"
      flame_props.SUCCESSFUL_INTERNET_TIMES_CONFIG = "GATOR.INTERNET_CHECK"
      flame_props.INTERNET_CHECK_KEY = "CONNECTION_TIME"
      flame_props.BPS_CONFIG = "GATOR.LEAK.BANDWIDTH_CALCULATOR.BPS_QUEUE"
      flame_props.BPS_KEY = "BPS"
      flame_props.PROXY_SERVER_KEY = "GATOR.PROXY_DATA.PROXY_SERVER"
      flame_props.getFlameId = function()
        if config.hasKey(flame_props.FLAME_ID_CONFIG_KEY) then
          local l_1_0 = config.get
          local l_1_1 = flame_props.FLAME_ID_CONFIG_KEY
          return l_1_0(l_1_1)
        end
        return nil
      end
    end
  end
end
```



Adobe Lightroom

more than a million lines of
Lua code on top of half a
million lines of C++/C/ObjC.



Embedded systems

TVs (Samsung), routers (Cisco), keyboards (Logitech), printers (Olivetti, Océ), car panels (Volvo, Mercedes), set-top boxes (Ginga, Verizon), M2M devices (Sierra Wireless, Koneki), calculators (TI-Nspire), mobiles (Huawei), ...



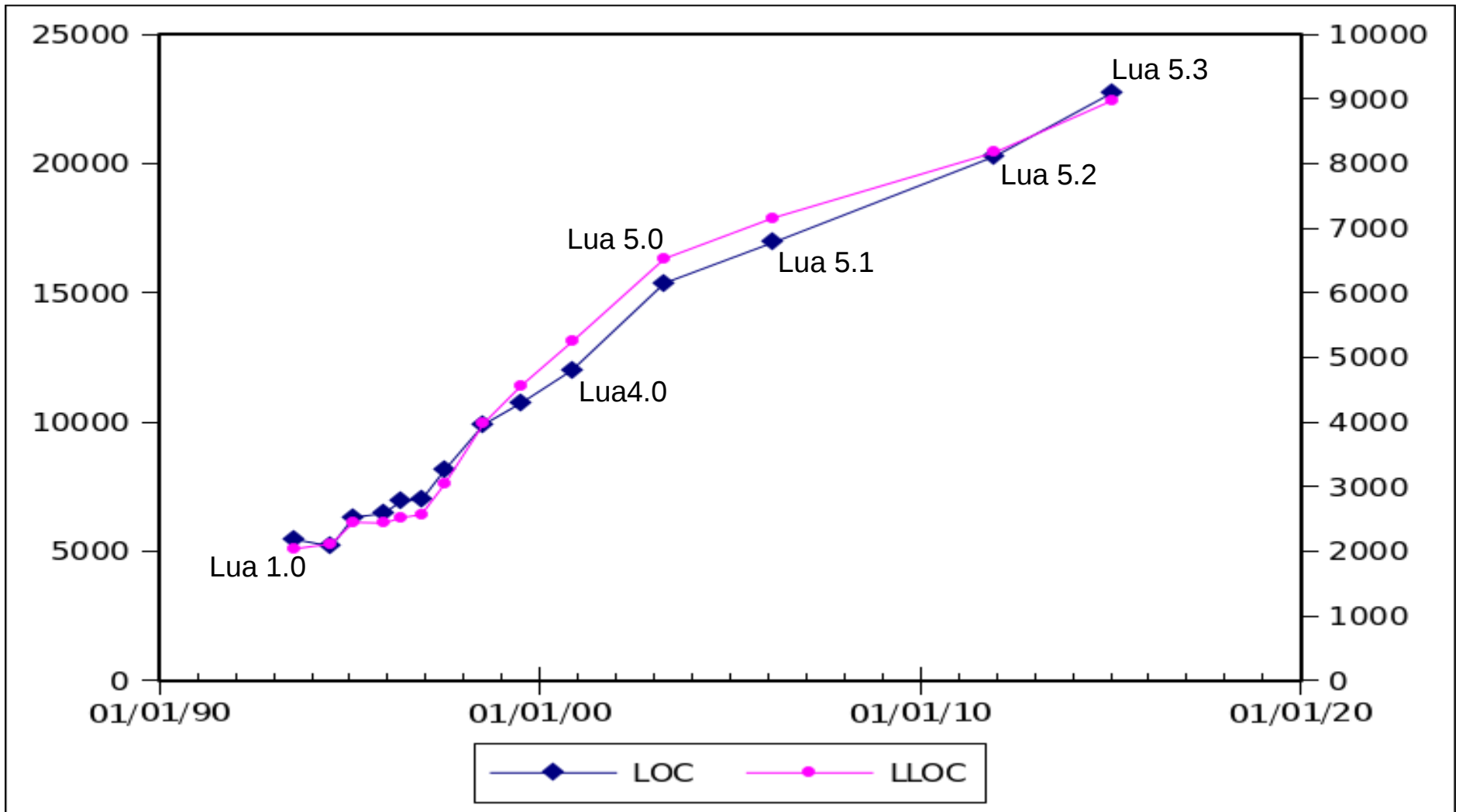
Why Lua?

- Portable
- Small
- Simple
- Emphasis on *scripting*

Portability

- Runs on most platforms we ever heard of.
 - Posix (Linux, BSD, etc.), OS X, Windows, Android, iOS, Arduino, Raspberry Pi, Symbian, Nintendo DS, PSP, PS3, IBM z/OS, etc.
 - written in ANSI C, few platform-specific `#ifdef`'s
- Runs inside OS kernels.
 - NetBSD, Linux
- Runs on the bare metal.
 - Arduino, AVR32, NodeMCU

Size



```
$ size /usr/bin/lua
  text    data    bss     dec     hex filename
188875   1344     40  190259  2e733 /usr/bin/lua
```

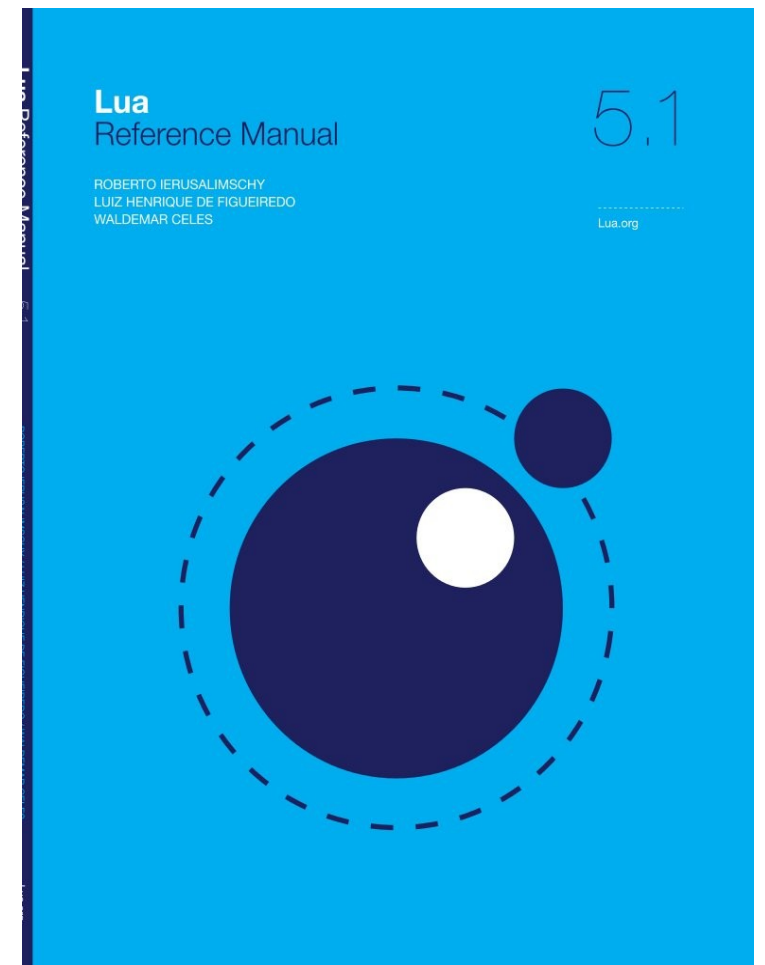
Simplicity

Reference manual with less than 100 pages
(proxy for complexity)

Documents the language, the
libraries, and the C API.

(spine)

Lua Reference Manual 5.1 ROBERTO IERUSALIMSKY / LUIZ HENRIQUE DE FIGUEIREDO / WALDEMAR CELES Lua.org



Lua and Scripting

- Lua is implemented as a library.
- Lua has been designed for scripting.
- Good for *embedding* and *extending*.
- Embedded in C/C++, Java, Fortran, C#, Perl, Ruby, Python, etc.

Aren't those goals about the implementation, not about the language?

Aren't these goals about the implementation, not about the language?

No.

Aren't these goals about the implementation, not about the language?

No.

(Maybe a little, but big impact on language design.)

Design Principles

- Few but powerful mechanisms.
 - associative arrays, functions, and coroutines
- Emphasis on the “eye of the needle”.
 - how to pass a mechanism through the Lua-C API



Tables

- Tables in Lua are associative arrays.
- All Lua types can be used as keys.
 - except nil
- Lua uses tables for all its data structures
 - records, lists, arrays and matrices (dense and sparse), sets, bags
- Tables implement all those data structures in *simple and efficient ways*.

Structures

```
a = {x = 10.5, y = -3.2}  
print(a.x)      --> 10.5  
print(a["x"])   --> 10.5  
a.z = 0.0
```

Short strings are automatically internalized, working like symbols in other languages.

Lists and Arrays

```
a = {}  
for i = 1, N do a[i] = f(i) end  
  
for i = 1, #a do  
    print(a[i])  
end
```

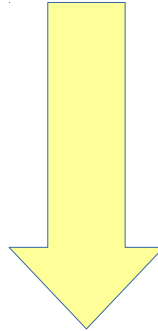
- Indices are integers, not strings.
- Lua automatically stores and indexes arrays as arrays (not as hash tables).
- Sparse arrays come for free:
 - `a[1e12] = 1` -- goes to the hash

“Closures”

- Anonymous functions as first-class values with lexical scoping.
- Now more common in non-functional languages, but not that common.
 - closing on variables x closing on values
 - other idiosyncrasies
- Few non-functional languages use closures as pervasively as Lua.

All functions in Lua are anonymous.

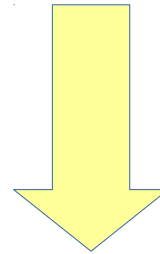
```
function add (x, y)
  return x + y
end
```



```
add = function (x, y)
  return x + y
end
```


All functions we write in Lua are nested.

```
load [[  
  function add (x, y)  
    return x + y  
  end  
  print(add(3, 5))  
]]
```



```
eval [[  
  return function ()  
    add = function (x, y)  
      return x + y  
    end  
    print(add(3, 5))  
  end  
]]
```

Modules

- Tables populated with functions


```
local math = require "math"  
print(math.sqrt(10))
```

- Several facilities come for free
 - submodules
 - local names

```
local m = require "math"  
print(m.sqrt(20))  
local f = m.sqrt  
print(f(10))
```

Objects

- first-class functions + tables \approx objects
- syntactical sugar for methods
 - handles *self*

`a:foo(x)`  `a.foo(a, x)`

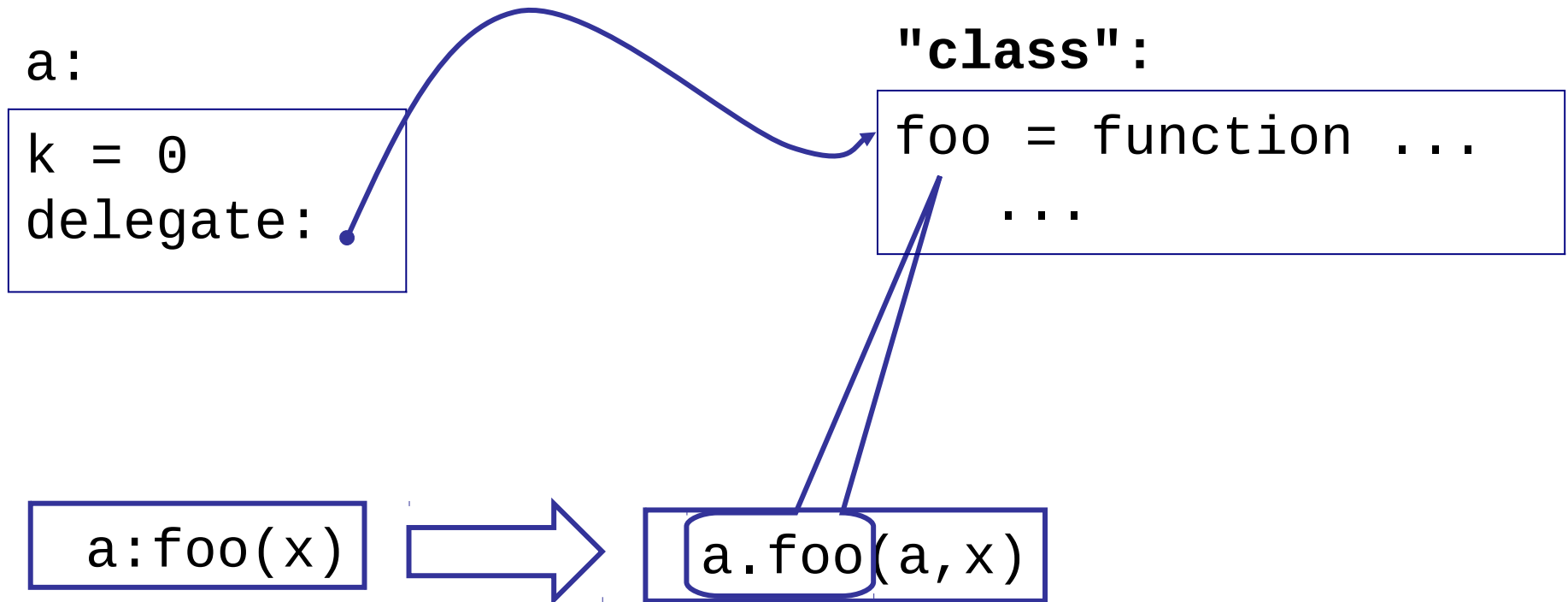
```
function a:foo (x)
  ...
end
```

```
a.foo = function (self, x)
  ...
end
```

Delegation

- Field-access delegation (instead of method-call delegation).
- When a delegates to b, an access to a field absent in a is delegated to b.
 - $a[k]$ becomes $(a[k] \text{ or } b[k])$
- Allows prototype-based and class-based objects.
- Allows single inheritance.

Delegation at work



Environments

- Globals are evil.
- Globals are handy:

```
> a = 3  
> print(a^12)
```

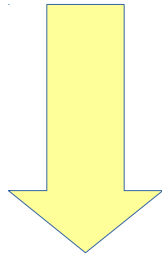
Environments

- Globals are evil.
- Globals are handy:

```
> a = 3  
> print(a^12)
```

Lua does not have global variables, but it goes to great lengths to pretend it does.

```
load([[
    a = 3
    print(a)
]],  $\beta$ )
```



```
eval[[
    local _ENV =  $\beta$ 
    return function (...)
        _ENV.a = 3
        _ENV.print(_ENV.a)
    end
]]
```

By default, β is a fixed table, which works like a global environment. But it can be anything.

Using Environments for Modularity

```
local math = require "math"
local foo = require "foo" }
```

import list

```
_ENV = <something>
```

```
-- From this point on, all accesses to  
-- free names are mediated by <something>
```

```
< your code >
```

Typical Uses

- `_ENV = nil`
 - no more access to globals. Any use of a free name will raise a run-time error.
- `_ENV = {}`
 - private environment. All accesses to free names go to the new table.
- `_ENV = setmetatable({}, {__index = _G})`
 - environment delegates to global table. Writes go to new table, reads may access global variables.

Coroutines

- Old and well-established concept, but with several variations.
- Variations are not equivalent:
 - several languages implement restricted forms of coroutines that are not equivalent to one-shot continuations

Coroutines in Lua

```
c = coroutine.create(function ()  
    print(1)  
    coroutine.yield()  
    print(2)  
end)  
  
coroutine.resume(c) --> 1  
coroutine.resume(c) --> 2
```

Coroutines in Lua

- First-class values
 - in particular, we may invoke a coroutine from any point in a program
- *Stackful*
 - a coroutine can yield anywhere in its execution
- Asymmetrical
 - different commands to resume and to yield

Coroutines in Lua

- Simple and efficient implementation.
 - the easy part of multithreading
- First class + stackful = complete coroutines.
 - equivalent to one-shot continuations
 - we can implement call/cc
- Coroutines present one-shot continuations in a format that is more familiar to most programmers.

Coroutines x Continuations

- Most uses of continuations can be coded with coroutines.
 - who has the main loop” problem
 - producer-consumer
 - extending x embedding
 - iterators x generators
 - collaborative multithreading
- Weaker than multi-shot continuations
 - e.g., oracle functions

Final Remarks

- Scripting is a relevant technique for any programmer's toolbox.
- Language interoperability is not an implementation detail.
- Lua has been designed for scripting.
 - tables, functions, and asymmetric coroutines are easy to map into an API
 - anything built on top of that is also easy to map