Lua 5.1 para Programadores

Renato Maia
maia@inf.puc-rio.br

Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática

28 de maio de 2009

O Que é Lua?

- Mais uma linguagem dinâmica.
  - Interpretação de código

```
code = loadstring("print('Hello, World!')")
code() --> Hello, World!
```

  - Tipagem dinâmica

```
a = 1
print(a+a) --> 2
a = "a"
print(a+a) --> attempt to perform arithmetic on global 'a'
```

  - Coleta automática de lixo

```
file = assert(io.open("file.txt", "w"))
file:write(a)
file:close()
file = nil -- conteudo de 'file' vira lixo a ser coletado
```

  - Reflexão computacional

```
function string:trim()
  return self:match("^%s*(.-)%s*$")
end
username = "  admin  "
print(username:trim()) --> admin
```

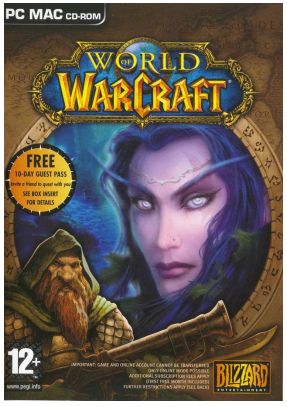
O Que é Lua?

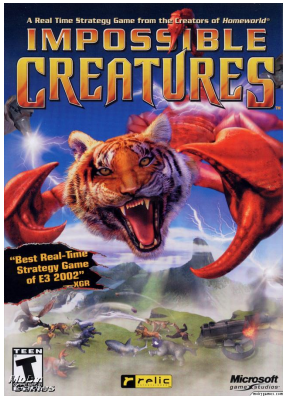
- Linguagem de extensão estensível.
  - Ênfase em comunicação inter-linguagens.
  - Enfatiza desenvolvimento em múltiplas linguagens.
- Uma biblioteca ANSI C:

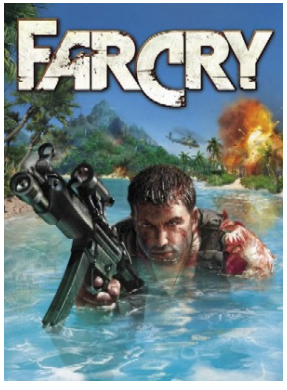
```c
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

int main(int argc, char* argv[])
{
  lua_State *L = lua_open();
  luaL_openlibs(L);
  luaL_dostring(L, "print('Hello, World!')");
  lua_close(L);
  return 0;
}
```

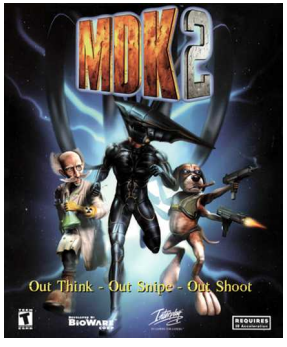
- Única linguagem fora do eixo EUA/Europa/Japão a ser adotada mundialmente (Ruby é a única do Japão).
  - É genuinamente brasileira (PUC-Rio 1993–2009).

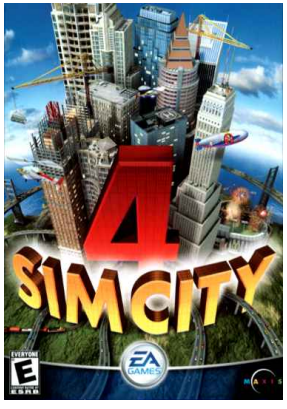
Onde Lua é Usada?

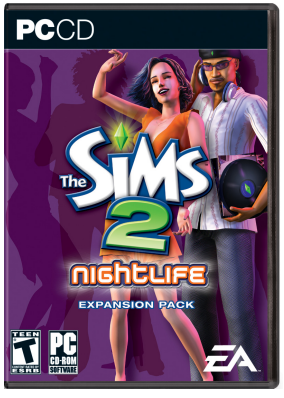


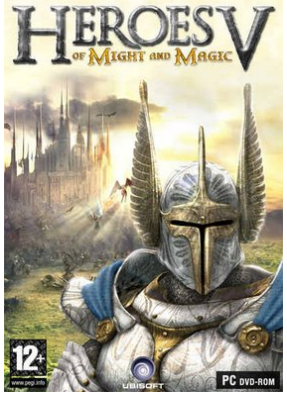


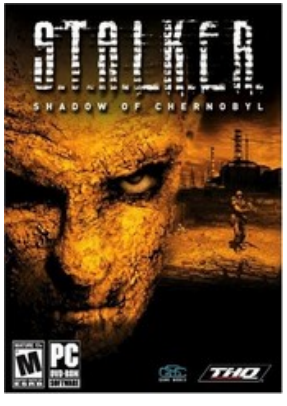












"It is easy to see why Lua is rapidly becoming the de facto standard for game scripting".

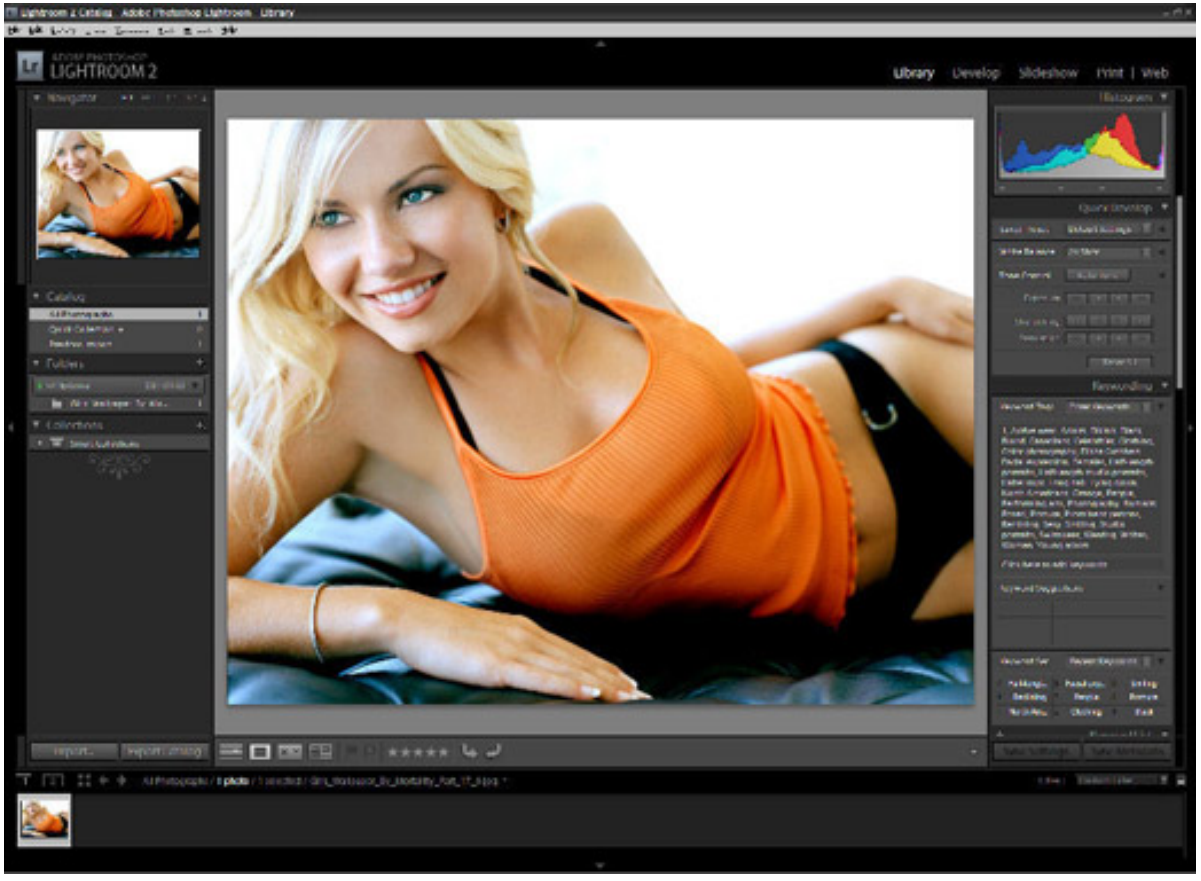
Artificial Intelligence for Games
Morgan Kaufmann
2006

"It's quite possible that game developers will look back at the 2000s as the decade of Lua".

Game Programming
Charles River Media
2005

Onde Lua é Usada?

- Adobe Photoshop Lightroom



"Over 40% of Adobe Lightroom is written in Lua."

"So what we do with Lua is essentially all of the application logic from running the UI to managing what we actually do in the database."

Mark Hamburg
Fundador do projeto Adobe Photoshop Lightroom

Onde Lua é Usada?

- *Firmware* de impressoras (Olivetty)
- Analisador de protocolos (Wireshark)
- Monitoramento remoto (Omnitronix)
- Pós-produção de filmes (Digital Fusion, eyeon)
- Servidores Web (RealTimeLogic)

Por Que o Sucesso de Lua?

Livre A licença de Lua é muito liberal (*MIT license*).

Portável Escrita em ANSI C ∩ ANSI C++, sem `ifdef`'s.

Embarcável Se integra facilmente com outras linguagens.

Pequena Fontes em torno de 17 mil linhas de código C. Interpretador+bibliotecas == 153KB (Linux).

Rápida *"Lua code tends to be executed much faster than other interpreted languages, so fast that 'as fast as Lua' has become a proverbial expression."*

`www.h3rald.com`

Poderosa, mas simples *"Lua gives you the power; you build the mechanisms."*

Programming in Lua
Roberto Ierusalimschy, 2003.

Como é Lua?

Tipos de Dados

nil	function
boolean	table
number	thread
string	userdata

Operadores

Aritméticos	+ - * / ^
Relacionais	== ~= < > <= >=
Lógicos	not and or
Concaten.	..
Comprim.	#

Estruturas de Controle

```
local e, b, c = 0, 0, 0
local ln = io.read()
while ln ~= nil do
  if ln == "" then
    e = e + 1
  elseif ln:match("^%s&") then
    b = b + 1
  else
    c = c + 1
  end
  repeat
    print(ln:sub(1, 80))
    ln = ln:sub(81)
  until ln == ""
  ln = io.read()
end
local t = e+b+c
for i=1, 80 do
  io.write(
    (i/80<c/t) and "." or " ")
end
```

Strings

- São imutáveis.

```
version = "Lua".." ".."5.1"
invert(version)
print(version) --> Lua 5.1
```

- Cada valor string é único.

```
if version == "Lua 5.1" then print("iguais") end --> iguais
```

- Porém há um custo em criar novas strings, mesmo uma substring. Em particular, o custo de concatenação é “alto”.

```
-- ruim
local text = ""
for line in io.lines() do
  text = text..line.."\n"
end

-- melhor
local text = io.read("*a")
```

- Alternativas: `table.concat`, `string.gsub`, etc.

Funções

- São valores de primeira-classe e anônimas.

```
print(string.gsub("Lua $MAJOR.$MINOR", "$(%u+)", function(name)
  if name == "MAJOR" then return 5 end
  if name == "MINOR" then return 1 end
end))                          --> Lua 5.1
```

- Não define os parâmetros que deve receber.

```
twoArgs = function(a, b) print(a, b) end
twoArgs(1, 2, 3)               --> 1    2
twoArgs(1)                     --> 1    nil
```

- Pode retornar um número arbitrário de parâmetros. (1)

```
function getNums(n)
  if n == 1 then return 1 end
  if n == 2 then return 1, 2 end
  if n == 3 then return 1, 2, 3 end
end
local a, b = getNums(3)
print(a, b)                    --> 1    2
local a, b = getNums(1)
print(a, b)                    --> 1    nil
print(getNums(3))              --> 1    2    3
```

Manipulando Argumentos Variados

- Dando diferentes nomes aos argumentos.

```
function process(message, ...)
  if message == "Request" then
    local object, operation, args = ...
    return dispatch(object, operation, args)
  elseif message == "Reply" then
    local success, result = ...
    if success then return result end
    error(result)
  end
end
```

- A função `select`.

```
function maximum(...)
  local max = -math.huge
  for i=1, select("#", ...) do
    local value = select(i, ...)
    max = value > max and value or max
  end
  return max
end
print(maximum(1, 2, 3)) --> 3
print(maximum(nil))     --> attempt to compare number with nil
```

Funções

- Podem ser recursivas e realizam chamadas finais próprias (*proper tail call*).

```
function pickOS()
  io.write("Pick an OS: MacOS, Linux, Windows\n>")
  local choice = io.read()
  if choice == "MacOS" then return pickMacOS() end
  if choice == "Linux" then return pickLinux() end
  if choice == "Windows" then return pickWindows() end
  return pickOS()
end

function pickMacOS()
  io.write("Pick an MacOS version: 1--10 or type 'Back'\n>")
  local choice = io.read()
  if choice == "Back" then return pickOS() end
  local version = tonumber(io.read())
  if version and version > 0 and version <= 10 then
    return setOS("MacOS", version)
  end
  return pickMacOS()
end
```

...

`for` Genérico

- Pode-se usar uma função para iterar num conjunto de elementos.

```
local function iterator(string, pos)
  pos = pos + 1
  if pos <= #string then
    return pos, string:sub(pos, pos)
  end
end

function allchars(string)
  return iterator, string, 0
end

for pos, char in allchars(version) do
  print(pos, char)
end
```

```
---> 1    L
---> 2    u
---> 3    a
---> 4
---> 5    5
---> 6    .
---> 7    1
```

Fechos de Função

- Funções são na realidade fechos que capturam variáveis do escopo em que são criadas

```
function compose(f, g)
  return function(...)
    return f(g(...))
  end
end

function normal(x, y, z)
  local len=(x^2+y^2+z^2)^.5
  return x/len, y/len, z/len
end

local f = compose(maximum,
                  normal)

print(f(1,2,3)) --> 0.8017...
```

```
function counter()
  local current = 0
  return function()
    current = current + 1
    return current
  end, function()
    current = current - 1
    return current
  end
end

local inc, dec = counter()
print(inc()) --> 1
print(inc()) --> 2
print(inc()) --> 3
print(dec()) --> 2
print(dec()) --> 1
print(dec()) --> 0
```

Interpretação

- A função `loadstring` permite criar uma função cujo corpo é o código Lua dado pela string passada por parâmetro.

```lua
local fat1 = loadstring([[
  local fat = ...
  for i=fat-1, 2, -1 do
    fat = fat*i
  end
  return fat
]])

print(fat1(5)) --> 120
```

```lua
local fat2 = function(...)
  local fat = ...
  for i=fat-1, 2, -1 do
    fat = fat*i
  end
  return fat
end

print(fat2(5)) --> 120
```

- As demais funções que interpretam código Lua funcionam de forma similar:
  - `loadfile(path):loadstring(io.open(path):read("*a"))`
  - `dofile(path):assert(loadfile(path))()`

Tabelas

- São vetores associativos, onde as chaves e valores podem ser qualquer valor diferente de `nil`.

```lua
local tab = {
  [123]      = "one two three",
  ["dois"]   = 2,
  ["dummy"]  = function()end,
  ["table"]  = {
    [function()end] = "function",
    [{}] = "table"
  },
}

tab["dois"] = "two"
print(tab["dois"]) --> two
print(tab[321])    --> nil
print(tab[nil])    --> nil

for key, value in pairs(tab) do  --> dummy  function: 0x12fdf0
  print(key, value)              --> dois   two
end                              --> 123    one two three
                                 --> table  table: 0x12fce0
```

Sequências

- Valores em índices inteiros não-negativos são armazenados de forma eficiente.

```
local list = { "a","b","c","d","e" }
list[2] = "B"
list[4] = "D"
for i = 1, #list do ---> a
  print(list[i])    ---> B
end                 ---> c
                    ---> D
                    ---> e
```

- Idiomas comuns na manipulação de listas.

```
local lines = {}
for line in io.lines() do
  lines[#lines+1] = line
end
io.write(table.concat(lines, "\n"))
```

Sequências Degeneradas

- Uma sequência só é “bem formada” se não tiver “buracos”.

```
local list = { "a",nil,"c",nil,"e" }
print(#list)                    --> 5
local list = { [1]="a",[2]=nil,[3]="c",[4]=nil,[5]="e" }
print(#list)                    --> 1
```

- Funções que manipulam sequências tem o mesmo comportamento.

```
local list = { "a","b","c", [24]="x",[25]="y",[26]="z" }
for index, value in ipairs(list) do
  list[index] = value..value
end
print(unpack(list))         --> aa      bb      cc
print(table.concat(list))   --> aabbcc
```

Conjuntos

- Elementos são armazenados como chave.

```
local primes = {}
for i=1, N do
  primes[i] = true
end

for i=2, N^.5 do
  if primes[i] ~= nil then
    for j=i+i, N, i do
      primes[j] = nil
    end
  end
end

print("Prime numbers smaller than "..N.." are:")
for prime in pairs(primes) do
  io.write(prime, ", ")
end
```

Registros

- Açúcar sintático para representar registros.

```
local contact = {
  name     = "Fulano de Tal",
  mail     = "fulano@mailcatch.com",
  phone    = 55212345678,
  fax      = 55213456789,
  webpage  = "http://www.blog.com/~fulano",
  address = {
    street     = "Rua Sem Fim",
    number     = 321,
    postalcode = 23456789,
    state      = "RJ",
    city       = "Rio de Janeiro",
    country    = "Brazil",
  },
}
print(contact.address.country)          --> Brazil
contact.address.country = "Brasil"
print(contact["address"]["country"])    --> Brasil
```

Misturando Usos Numa Mesma Tabela

- Sequências Esparsas

```
local sparse = { "a", "b", [10] = "j", [11] = "k", n = 26 }
```

- Conjunto Ordenado

```
local letters = {}
for i=1, 10 do
  local letter = string.char(math.random(65, 90))
  letters[#letters+1] = letter
  letters[letter] = #letters
end
print("Type a letter (A–Z)")
local index = letters[io.read():upper()]
if index ~= nil then
  print("Your letter was the "..index.."th random letter")
else
  print("Your letter was not generated. The letters were:")
  print(table.concat(letters, " "))
end
```

Ambientes

- Variáveis globais são na realidade campos da tabela _G.

```
assert(_G["print"] == print)
assert(_G.loadstring == loadstring)
```

- Cada função pode ter um ambiente global diferente.

```
local sandbox = {
  -- include only safe functions
   print = print,
}
local code = assert(loadstring([[
 print = nil
 os.execute('sudo rm -fR .')
]]))
setfenv(code, sandbox)
local success, errmsg = pcall(code)
if not success then
   print(errmsg) --> [string " print = nil..."]:2: attempt to in
end
```

Módulos

- Funções em Lua são geralmente organizadas em módulos que são tabelas que definem um espaço de nomes separado.

```
print(type(table))          --> table
print(type(table.concat)) --> function
```

- A função `module` é fornecida para facilitar a criação de módulos em Lua.

```
local select = select

module("table.util")

function newset(...)
  local set = {}
  for i=1,select("#", ...) do
    set[select(i, ...)]=true
  end
  return set
end
```

```
require("table.util")

s = table.util.newset(2,3,5,7)
print(s[1]) --> nil
print(s[2]) --> true
print(s[3]) --> true
```

Objetos

- Objetos são tabelas contendo funções (operações) e outros valores (atributos).

```
local Q = {
  head = 0,
  tail = 0,
}
function Q:enqueue(val)     --=> Q.enqueue = function(self, val)
  self.tail = self.tail + 1
  self[self.tail] = val
end
function Q:dequeue()        --=> Q.dequeue = function(self)
  if self.head < self.tail then
    self.head = self.head + 1
    return self[self.head]
  end
end

Q:enqueue("first")          --=> Q.enqueue(Q, "first")
Q:enqueue("second")
Q:enqueue("third")
print(Q:dequeue())          --> first
print(Q:dequeue())          --> second
```

Construtores

- Uso de funções para validar estruturas.

```
function Book(info)
  assert(type(info) == "table", "invalid book entry")
  assert(type(info.title) == "string", "invalid book title")
  assert(type(info.author) == "string", "invalid book author")
  assert(type(info.year) == "number", "invalid book year")
  if info.edition == nil then
    info.edition = 1
  else
    assert(type(info.edition) == "number", "invalid book editio
  end
  info.entry = "book"
  return info
end

local PiL = Book{
  title   = "Programming in Lua",
  author  = "Roberto Ierusalimschy",
  edition = 2,
  year    = 2006,
}
```

Construtores

- Uso de funções para construir objetos.

```
function Queue()
  local data = {}
  local head = 0
  local tail = 0
  return {
    enqueue = function(val)
      tail = tail + 1
      data[tail] = val
    end,
    dequeue = function()
      if head < tail then
        head = head + 1
        return data[head]
      end
    end,
  }
end
```

```
local Q = Queue()

Q.enqueue("first")
Q.enqueue("second")
print(Q.dequeue()) --> first
print(Q.dequeue()) --> second

print(Q.head)      --> nil
print(Q.tail)      --> nil
print(Q.data)      --> nil
```

Construtores

- Uso de funções para construir objetos.

```lua
function Queue()
  local data = {}
  local head = 0
  local tail = 0
  return {
    enqueue = function(val)
      tail = tail + 1
      data[tail] = val
    end,
    dequeue = function()
      if head < tail then
        head = head + 1
        return data[head]
      end
    end,
  }
end

local Q = Queue()
Q.enqueue("first")
print(Q.head) --> nil
```

```lua
local function enqueue(self,val)
  self.tail = self.tail + 1
  self.data[self.tail] = val
end
local function dequeue(self)
  if self.head < self.tail then
    self.head = self.head + 1
    return self.data[self.head]
  end
end
function Queue()
  return {
    head=0, tail=0, data={},
    enqueue = enqueue,
    dequeue = dequeue,
  }
end

local Q = Queue()
Q:enqueue("first")
print(Q.head) --> 0
```

Meta-tabelas

- São tabelas contendo campos especiais que estendem o comportamento de valores em Lua.

```
function Memoize(func)
  local metatable = {
    __index = function(self, key)
      local value = func(key)
      self[key] = value
      return value
    end,
  }
  local table = {}
  setmetatable(table, metatable)
  return table
end

local sqrt = Memoize(math.sqrt)
print(sqrt[121])  --> 11
print(sqrt[1024]) --> 32
```

Prototipação

- Compartilhando membros de outro objeto.

```
John = { name = "John Doe" }
function John:show()
  print("Hi, I'm a "..self.name)
end
John:show()                  --> Hi, I'm a John Doe

function clone(proto)
  local ProtoLike = { __index = proto }
  return setmetatable({}, ProtoLike)
end

guy = clone(John)
guy.name = "guy like John"
guy:show()                   --> Hi, I'm a guy like John
```

Classes

- Fica como exercício. ☺
  - Como definir um conjunto de membros que deve ser compartilhado por todos os objetos de uma classe?
  - Como definir uma forma de criar objetos iniciando seus membros?
  - Como criar hierarquias de classe de forma que uma herde (compartilhe) os membros definidos por outra?

Co-Rotinas

- É uma linha de execução com seu próprio contexto (*e.g.* contador de programa, pilha, variáveis locais, etc.).

```
local co = coroutine.create(function(start)
  local current = start
  local command
  repeat
    command, arg = coroutine.yield(current)
    if      command == "+" then current = current + arg
    elseif command == "-" then current = current - arg
    elseif command ~= "=" then error("bad op")
    end
  until command == nil
  return current
end)

print(coroutine.resume(co, 5))     --> true   5
print(coroutine.resume(co,"+",2)) --> true   7
print(coroutine.resume(co,"+",3)) --> true   10
print(coroutine.resume(co,"-",5)) --> true   5
print(coroutine.resume(co,"*",2)) --> false  ...:9: bad op
print(coroutine.resume(co,"="))   --> false  cannot resume dead
```

Co-Rotinas como Iteradores

- Gedar todas as permutações de uma sequência.

```
local function permgen(a, n)
  if n == 0 then
    -- produz uma permutação
    coroutine.yield(a)
  else
    for i=1,n do
      a[n],a[i] = a[i],a[n]
      permgen(a, n-1)
      a[n],a[i] = a[i],a[n]
    end
  end
end

function perm(a)
  return coroutine.wrap(function()
    permgen(a, #a)
  end)
end
```

```
for p in perm({"a","b","c"}) do
  print(unpack(p))
end
```

```
--> b   c   a
--> c   b   a
--> c   a   b
--> a   c   b
--> b   a   c
--> a   b   c
```

Considerações Finais

- O que atrai os usuários atuais de Lua:
  - Alta portabilidade.
  - Eficiência e tamanho.
  - Facilidade para descrição de dados.
  - Integração fácil com outras linguagens.
  - Extensibilidade.
- Outros pontos marcantes de Lua.
  - Implementação da máquina virtual.
  - Implementação de JIT.
  - Resgate do conceito de co-rotinas.
- Comunidade de Lua.
  - Relativamente pequena, mas crescendo.
  - Bastante técnica.
  - Muito amigável.

Fim

http://www.lua.org/

Classes

- Há muitos modelos e formas diferentes de fazer. Um ex.:

```
function class(members)
  members.__index = members
  return members
end


JohnClass = class(John)

function JohnClass:__tostring()
  return self.name
end


myJohn = setmetatable({}, JohnClass)
myJohn:show()                  --> Hi, I'm a John Doe
print(myJohn)                  --> John Doe
```

Classes

- E herança simples?

```
BigJohnClass = class(clone(JohnClass))

function BigJohnClass:show()
  print("Hi, I'm a big "..self.name)
end


myBigJohn = setmetatable({}, BigJohnClass)
myBigJohn:show()              --> Hi, I'm a big John Doe
print(myBigJohn)              --> table: 0x102ee0
```

- E herança múltipla?
  - Fica como exercício. ☺
  - Dica: use uma função no campo `__index`.

Classes

- E o construtor?

```
function new(class, ...)
  local object = setmetatable({}, class)
  if type(object.init) == "function" then
    object:init(...)
  end
  return object
end

NamedJohnClass = class(clone(JohnClass))
function NamedJohnClass:init(name)
  self.name = "John "..name
end

myJohnSmith = new(NamedJohnClass, "Smith")
myJohnSmith:show()                    --> Hi, I'm a John Smith
```

Tabelas Fracas

- Atributos privados.

```
WeakKeys = { __mode = "k" }
SecretAgentJohnClass = class(clone(JohnClass))
do
  local AliasOf = new(WeakKeys)
  local SecretOf = new(WeakKeys)
  function SecretAgentJohnClass:init(alias, secret)
    AliasOf[self] = alias
    SecretOf[self] = secret
  end
  function SecretAgentJohnClass:tellme(name)
    if name == AliasOf[self] then return SecretOf[self]
    elseif name == self.name then return "I'm a "..self.name
    else return "I'm not "..name
    end
  end
end

myAgent = new(SecretAgentJohnClass, "Bond, John Bond",
                "Lua is as much OO as you want it to be")
print(myAgent:tellme("John Doe"))          --> I'm a John Doe
print(myAgent:tellme("Bond, John Bond")) --> Lua is as much OO
```

Multithread Cooperativo

```
local threads = {
  coroutine.create(function()
    for i=1, 3 do
      threads[#threads+1] = coroutine.create(function()
        for step=1, 3 do
          print(string.format("%s: step %d of 3", name..i, step))
          coroutine.yield()
        end
      end)
    end
  end),
}
while #threads > 0 do
  local i = 1
  repeat
    local thread = threads[i]
    coroutine.resume(thread)
    if coroutine.status(thread) == "dead" then
      table.remove(threads, i)
    else
      i = i + 1
    end
  until i > #threads
end
```