



material adaptado de:

Uma Introdução à Programação em Lua

Roberto Ierusalimschy

CSBC 2012

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO



O Que é Lua

- Mais uma linguagem dinâmica
 - alguma similariedade com Perl, Python, ...
- Uma linguagem de descrição de dados
 - anterior a XML
- Ênfase em scripting
 - ≠ linguagem dinâmica!
 - ênfase em comunicação inter-linguagens
 - enfatiza desenvolvimento em múltiplas linguagens

Dinamismo

- interpretação dinâmica de código
 - linguagem capaz de interpretar código criado dinamicamente no mesmo ambiente de execução do programa (loadstring)
- tipagem dinâmica forte
 - tipagem dinâmica: verificação em tempo de execução
 - tipagem forte: linguagem jamais aplica uma operação a um tipo incorreto
- gerência automática de memória
 - coleta de lixo

Portabilidade

- Roda em praticamente todas as plataformas que já ouvimos falar
 - Unix, Windows, Windows CE, Symbian, BREW, hardware dedicado, Palm, PSP, etc.
- Escrita em ANSI C \cap ANSI C++
 - evita `#ifdefs`
 - evita pontos obscuros do padrão
- Núcleo é praticamente uma aplicação *free-standing*

Acoplabilidade

- Lua é uma biblioteca C
- API simples e bem definida
 - tipos simples
 - operações primitivas
 - modelo de pilha
- Bi-direcional
- Acoplada em C/C++, Java, Fortran, C#, Perl, Ruby, Ada, etc.

Como usar Lua

- uma única implementação principal, com diversas distribuições
- *stand alone* x embutida em algum programa
- para Windows, *Lua for Windows* vem se firmando como principal instalação
- para Linux, maioria das distribuições oferecem pacotes prontos

Lua stand alone

- quatro maneiras de executar um "programa"

```
$ lua -e "print(2^0.5)"
```

```
$ lua nome-do-arquivo
```

```
$ lua  
> print(2^0.5)
```

```
$ lua  
> dofile("nome-do-arquivo")
```

Os tipos

- number
- string
- boolean
- nil
- function
- table
- thread
- userdata

Number

- um único tipo numérico
 - a partir de Lua5.3: double ou int
- exatidão e eficiência em máquinas modernas

```
print(2^0.5)          --> 1.4142135623731
print(math.pi%0.01)  --> 0.0015926535897931
```

Boolean

- sem exclusividade em testes
- operadores booleanos operam sobre todos os tipos
- `nil` e `false` testam negativo

```
print(0 or 6)      --> 6
print(nil or 10)  --> 10
print(x or 1)
print(x > y and x or y)
```

Alguns exemplos simples

Soma dos elementos de um array

```
function add (a)
  local sum = 0
  for i = 1, #a do sum = sum + a[i] end
  return sum
end
```

```
print(add({10, 20, 30.5, -9.8}))
```



construtor

Soma dos elementos de um array

tamanho do array

```
function add (a)
  local sum = 0
  for i = 1, #a do sum = sum + a[i] end
  return sum
end
```

```
print(add({10, 20, 30.5, -9.8}))
```

Soma das linhas de um arquivo

```
function addfile (filename)
    local sum = 0
    for line in io.lines(filename) do
        sum = sum + tonumber(line)
    end
    return sum
end
```

Soma das linhas de um arquivo

```
function addfile (filename)
    local sum = 0
    for line in io.lines(filename) do
        sum = sum + tonumber(line)
    end
    return sum
end
```

iterador genérico



io

io.read – padrões:

"a" reads the whole file

"l" reads the next line (without newline)

"L" reads the next line (with newline)

"n" reads a number

num reads a string with up to *num* characters

Funções em Lua

- funções em Lua são valores dinâmicos de primeira classe

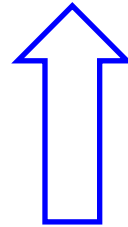
```
(function (a,b) print(a+b) end)(10, 20)
```

```
table.sort(t, function (a,b)  
    return a.key < b.key  
end)
```

Funções "convencionais"

- podemos armazenar funções em variáveis

```
inc = function (a) return a + 1 end
```



```
function inc (a)  
  return a + 1  
end
```

Múltiplos retornos

- funções em Lua podem retornar múltiplos valores

```
function foo (x) return x, x+1 end

print(foo(3))      --> 3    4

a, b = foo(45)
print(b, a)       --> 46, 45

ajuste de número de valores!

a = foo(45)
print(b, a)       --> 46, nil
a, b , c = foo(45)
print(c)          --> nil
```

Múltipla atribuição

Todos os prefixos de uma string

```
function prefixes (s, len)
  len = len or 0
  if len <= #s then
    return string.sub(s, 1, len),
           prefixes(s, len + 1)
  end
end
```

```
print(prefixes("alo")) --> a   al   alo
t = {prefixes("vazavaza")}
```

Todos os prefixos de uma string

```
function prefixes (s, len)
  len = len or 0
  if len <= #s then
    return string.sub(s, 1, len),
           prefixes(s, len + 1)
  end
end
```

```
print(prefixes("alo")) --> a al alo
```

Todos os prefixos de uma string

```
function prefixes (s, len)
  len = len or 0
  if len <= #s then
    return string.sub(s, 1, len),
           prefixes(s, len + 1)
  end
end
```

```
print(prefixes("alo", 2) ) → al  alo
```

Funções e escopo léxico

Regiões geométricas

- podemos desenvolver complexas estruturas de dados para representar regiões geométricas de forma geral
- ou podemos representar uma região geométrica diretamente por meio de sua função característica!

```
function C1 (x, y)
    return (x - 1.0)^2 + (y - 3.0)^2 <= 4.5^2
end
```

Regiões geométricas

- a função abaixo cria regiões circulares:

```
function circle (cx, cy, r)
    return function (x, y)
        return (x - cx)^2 + (y - cy)^2 <= r^2
    end
end
```

```
c1 = circle(5.0, -3.2, 4.5)
c2 = circle(0, 0, 1)
```

Combinando regiões

```
function union (r1, r2)
  return function (x, y)
    return r1(x, y) or r2(x, y)
  end
end
```

```
function inter (r1, r2)
  return function (x, y)
    return r1(x, y) and r2(x, y)
  end
end
```

Tabelas em Lua

- único mecanismo para estruturação de dados
- arrays associativos
 - associa chaves com valores
 - tanto chaves quanto valores podem ter qualquer tipo
- implementam estruturas de dados como arrays, estruturas (registros), conjuntos e listas
- e também objetos, classes e módulos

Construtores

- criação e inicialização de tabelas

```
{  
  {x = 5, y = 10}  
  {"Sun", "Mon", "Tue"}  
  {[exp1] = exp2, [exp3] = exp4}
```

Todos os prefixos de uma string

```
function prefixes (s, len)
  len = len or 0
  if len <= #s then
    return string.sub(s, 1, len),
           prefixes(s, len + 1)
  end
end
```

```
print(prefixes("alo")) --> a    al    alo
t = {prefixes("vazavaza")}
```

Estruturas

- nomes dos campos como chaves
- açúcar sintático **t.x** para **t["x"]**:

```
t = {z = 30}
t.x = 10
t.y = 20
print(t.x, t.y, t.z)
print(t["x"], t["y"], t["z"])
```

Estruturas de Dados (2)

- Arrays: inteiros como índices

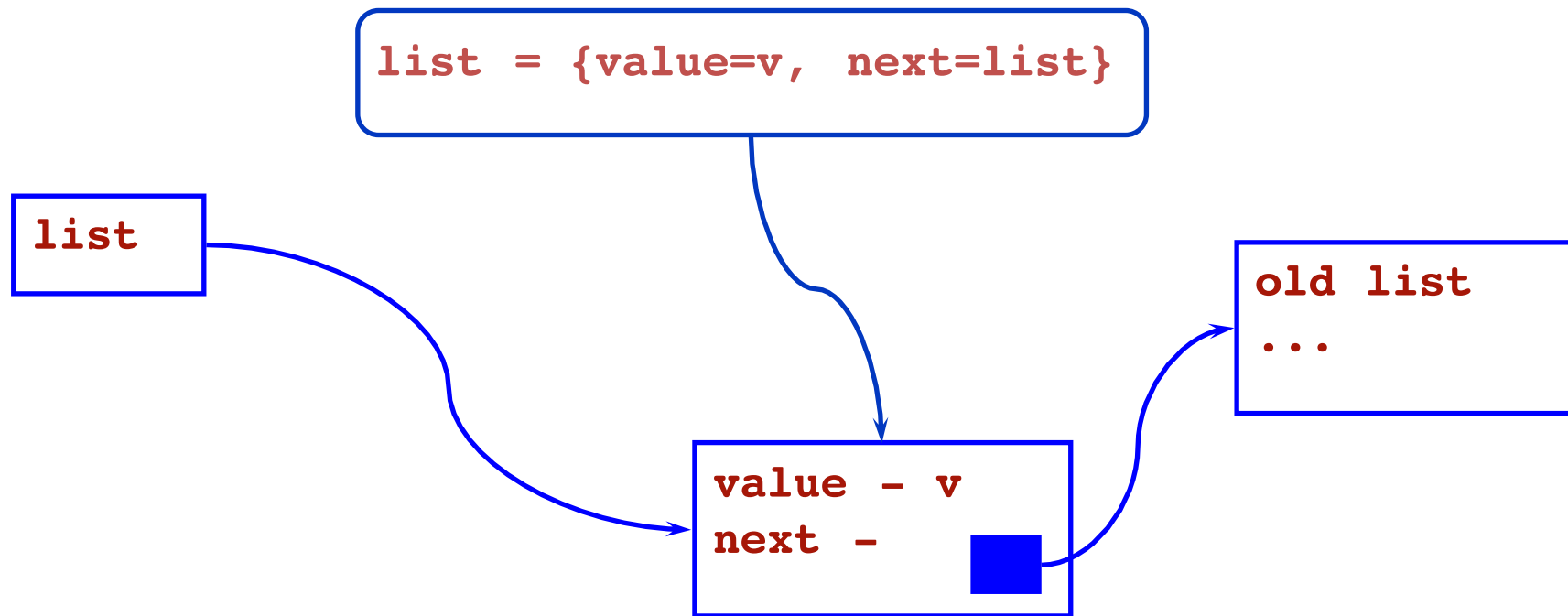
```
a = {}  
for i=1,n do a[i] = 0 end  
print(#a)
```

- Conjuntos: elementos como índices

```
t = {}  
t[x] = true      -- t = t U {x}  
if t[x] then    -- x ∈ t?  
    ...  
end
```


Listas Encadeadas

- Tabelas são *objetos*, criados dinamicamente



Exemplo: palavras mais frequentes

```
-- lê arquivo
local t = io.read("*all")

-- coleta e conta palavras
local count = {}
for w in string.gmatch(t, "%w+") do
    count[w] = (count[w] or 0) + 1
end

...
```

(ou se arquivo for muito grande)

```
-- lê, coleta e conta palavras
local count = {}
for line in io.lines() do
    for w in string.gmatch(line, "%w+") do
        count[w] = (count[w] or 0) + 1
    end
end
end

...
```

```
-- lista de palavras (para ordenar)
local words = {}
for w in pairs(count) do
    words[#words + 1] = w
end

-- ordena lista
table.sort(words, function (a,b)
    return count[a] > count[b]
end)

-- imprime as mais frequentes
for i=1, (arg[1] or 10) do
    print(words[i], count[words[i]])
end
```

tabelas

- indexamento por tipos arbitrários
 - podemos usar uma tabela indexada por sockets

percurso de tabelas

- arrays: ipairs
- outras: pairs

```
nt = {}  
for k, v in pairs(t) do  
    nt[v] = k  
end
```

Objetos

- funções de 1ª classe + tabelas ≈ objetos

```
Rectangle = {w = 100, h = 250}

function Rectangle.area ()
    return Rectangle.w * Rectangle.h
end
```

```
function Rectangle.area (self)
    return self.w * self.h
end
```

Chamada de métodos

- açúcar sintático para métodos
 - cuida de *self*

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



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

```
a:foo(x)
```



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


servidor iterativo em Lua

```
-- load namespace
local socket = require("socket")
-- create a TCP socket and bind it to the local host, at any port
local server = assert(socket.bind("*", 0))
-- find out which port the OS chose for us
local ip, port = server:getsockname()
-- print a message informing what's up
print("Please telnet to localhost on port " .. port)
print("After connecting, you have 10s to enter a line to be echoed")
-- loop forever waiting for clients
while 1 do
    -- wait for a connection from any client
    local client = server:accept()
    -- make sure we don't block waiting for this client's line
    client:settimeout(10)
    -- receive the line
    local line, err = client:receive()
    -- if there was no error, send it back to the client
    if not err then client:send(line .. "\n") end
    -- done with client, close the object
    client:close()
end
```

Fçs com # variável de argumentos

```
function foo (...)  
  -- ... is called the "vararg" expression  
  local sum = 0  
  for _, v in ipairs {...} do  
    sum = sum + v  
  end  
  return sum  
end
```

Lua como API

- Lua é organizada como uma biblioteca em C, não como um programa
- exporta pouco menos de 100 funções
 - executar trechos de código Lua, chamar funções, registrar funções C para serem chamadas por Lua, manipular tabelas, etc.
- O programa `lua` é um pequeno programa cliente da biblioteca Lua
 - menos de 400 linhas de código

Referências

- Roberto Ierusalimschy. Uma Introdução à Programação em Lua. CSBC 2012 (JAI).
- Roberto Ierusalimschy. Programming in Lua. 4th ed. Lua.org. 2016.
 - 3a edição traduzida para português
 - 1a edição disponível online
- www.lua.org