Descobrindo Lua

Sérgio Queiroz de Medeiros

smedeiros@inf.puc-rio.br
LabLua

Campus Party 2009

Instalando o Interpretador

- Para baixar o fonte:
  www.lua.org
- Para obter os arquivos binários:
  http://luaforge.net/projects/luabinaries/
- Para obter uma distribuição com "baterias" para o Windows:
  http://luaforwindows.luaforge.net/
- Se você está no Ubuntu, tente:
  sudo apt-get install lua5.1
  sudo apt-get install liblua5.1.0-dev (Para a API C)

Compilando o Interpretador

- Descompacte o código fonte de Lua e entre no diretório correspondente;
- No Linux, digite: make linux
- Para testar: make test
- Para instalar: make linux install
- Se não conseguir compilar Lua, provavelmente falta instalar libncurses.

Primeiros Passos

- Um pedaço de código Lua (um arquivo ou uma linha digitada no modo interativo) é chamado de trecho;
- Ponto e vírgula é opcional no fim da linha:

```
a = 1;
b = 2
a = 1 ; b = 2
a = 33  b = 44
```

- Variáveis são globais por padrão.

Primeiro Programa

```
01  function odd (n)
02    if n == 0 then
03      return false
04    else
05      return even (n-1)
06    end
07  end
08
09  function even (n)
10    if n == 0 then
11      return true
12    else
13      return odd (n-1)
14    end
15  end
16
17  local n = io.read ("*number")
18  print (even (n))
```

Operador de Módulo

- Lua também possui um operador de módulo %
- Uma forma mais simples de escrever o programa anterior seria:

```
01  function odd (n)
02    return n % 2 == 1
03  end
04
05  function even (n)
06    return n % 2 == 0
07  end
08
09  local n = io.read ("*number")
10  print (even (n))
```

Convenções Léxicas

- Lua é *case sensitive*:
  - bola;
  - BOLA;
  - bOIA.
- `--`

  indica um comentário de uma linha;
- `--[[`

  abre um comentário que termina com um

  `]]`

Tipos e Valores

- Não há definição de tipos em Lua;
- Existem oito tipos básicos:
  - nil;
  - boolean;
  - number;
  - function;
  - string;
  - userdata;
  - function;
  - thread;
  - table.
- Variáveis podem conter valores de qualquer tipo.

Exercitando

```
--usamos --> para indicar a saída do comando

print (type (a))     --> nil
a = 10
print (type (a))     --> number
a = "uma cadeia"
print (type (a))     --> string
a = print
a (type (a))         --> function
```

Tipos e Valores

- nil é diferente de qualquer outro valor e também é usado para indicar ausência de um valor;
- false e true são valores do tipo *Boolean*;
- Lua considera nil e false como um valor falso e qualquer outro valor como verdadeiro.

Números

- O tipo *Number* representa números reais (não há um tipo inteiro);
- Números são representados com precisão dupla (*double*);
- É fácil compilar Lua para usar precisão simples ou valores inteiros como tipo numérico básico.

Cadeias de Caracteres (*Strings*)

- São imutáveis:

```
a = "uma cadeia"
b = string.gsub (a,"uma","outra") -- muda a cadeia "a"
print (a)         --> uma cadeia
print (b)         --> outra cadeia
print (a .. b)    --> uma cadeiaoutra cadeia
```

- São delimitadas por aspas simples ou duplas:

```
a =  "um texto"
b = 'outro texto'
```

Operadores Relacionais

- Operadores relacionais de Lua:

  `<` `>` `<=` `>=` `==` `~=`

- Um objeto (*function*, *userdata*, *thread*, *table*) somente é igual a ele mesmo:

```
local tab1 = {}       -- cria uma tabela
local tab2 = {}

tab1.x = 33
tab2.x = 33
print (tab1 == tab2) --> false

tab2 = tab1
print (tab1 == tab2) --> true
tab2.x = 20
print (tab1.x)        --> 20
```

Operadores Lógicos

```
a = 3
b = nil

print (a or b)      --> 3

print (a and b)     --> nil

print (not a)       --> false

print (not 0)       --> false

print (not b)       --> true
```

Operadores Lógicos

- Uso do operador or para inicializar variáveis:

```
function initX (v)
  x = v or 100
end
```

- O operador ternário de C pode ser simulado com a expressão x and y or z:

```
s = (n % 2 == 0) and "par" or "ímpar"
```

Variáveis

- Um identificador pode ser qualquer cadeia (*string*) de letras, dígitos e sublinhados que não começa com um dígito:

```
bola
_bola
_bola_2008
declarando_uma_variavel_com_um_nome_bem_grande
```

- Lua usa identificadores começando com um sublinhado seguido por letras maiúsculas para alguns propósitos especiais:

```
_VERSION
_G
```

Variáveis

- Quando não atribuímos um valor a uma variável, ela é inicializada com nil

```
x = 1                   -- x recebe 1
b, c = "bola", 3        -- b recebe "bola" e c o valor 3
y                       -- o valor de y é nil
print (b, y)            --> bola   nil
```

Atribuição

- O número de variáveis no lado direito de uma atribuição não precisa ser igual ao número de valores no lado esquerdo:

```
a, b, sobrei = 1, 2
print (a, b, sobrei)  --> 1   2   nil

x, y = "bola", "casa", "sobrei"
print (x, y)             --> bola   casa

x, y = y, x      -- faz a troca de valores
print (x, y)     --> casa   bola
```

Escopo

- Uma variável declarada fora de qualquer bloco é visível em todo o arquivo
- Usamos do-end ou estruturas de controle para criar blocos:

```
local x, y = 33
print (x, y)     --> 33   nil
if x > 10 then
  local x = 5    -- alterando um "x" local
  y = 9
  print (x, y)   --> 5   9
else
  x = 2          -- alterando o "x" mais externo
  print (x, y)   --> 2   nil
end
print (x, y)     --> 33   9
```

Escopo

- Acesso a variáveis locais é mais rápido
- É comum salvar em variáveis locais valores que estão em variáveis globais:

```
local sqrt = math.sqrt
local bola = bola
local print = _G.print
```

Tabelas

- Único mecanismo de estruturação de dados em Lua;
- Implementa arrays associativos;
- Tabelas também são usadas para representar módulos, pacotes e objetos.

Tabelas

```
a = {}  --cria uma tabela e armazena referência em 'a'
```

Tabelas

```
a = {}  --cria uma tabela e armazena referência em 'a'

k = "x"
a[k] = 10       --nova entrada, chave="x" e valor=10
```

Tabelas

```
a = {}  --cria uma tabela e armazena referência em 'a'

k = "x"
a[k] = 10       --nova entrada, chave="x" e valor=10

a[20] = "Lua"  --nova entrada, chave=20 e valor="Lua"
```

Tabelas

```
a = {}  --cria uma tabela e armazena referência em 'a'

k = "x"
a[k] = 10       --nova entrada, chave="x" e valor=10

a[20] = "Lua"   --nova entrada, chave=20 e valor="Lua"

print (a["x"])          --> 10
k = 20
print (a[k])            --> Lua
```

Tabelas

```
a = {}  --cria uma tabela e armazena referência em 'a'

k = "x"
a[k] = 10       --nova entrada, chave="x" e valor=10

a[20] = "Lua"   --nova entrada, chave=20 e valor="Lua"

print (a["x"])          --> 10
k = 20
print (a[k])            --> Lua

a["x"] = a["x"] + 1   --   incrementa entrada "x"
print (a["x"])          --> 11
a.x = 42                --   mesmo que a["x"]
print(a.x,a["x"])       --> 42      42
```

Construtores de Tabelas

- Tabelas podem ser inicializadas como arrays:

```
dias = {"Domingo", "Segunda", "Terça", "Quarta",
        "Quinta", "Sexta", "Sábado"}

print (dias [4])   --> Quarta
```

Construtores de Tabelas

- Tabelas podem ser inicializadas como estruturas:

```
ponto = { x=10, y=20 }
```

- Equivalente a:

```
ponto = {}
ponto.x = 10
ponto.y = 20
```

Inserindo elementos

- Podemos inserir elementos usando a função table.insert:

```
local t = {10, 20, 30}

table.insert (t, 2, 15)   -- Insere 15 na posição 2

-- Agora temos t[1] = 10, t[2] = 15,
--             t[3] = 20 e t[4] = 30

table.insert (t, 35)      -- Insere 35 no fim

print (t[1], t[3], t[5])  --> 10   20   35
```

Tamanho da Tabela

- Se a tabela representa um array, podemos usar o operador de comprimento #:

```
local t = {"a", "b", "c"}
print (#t)                          --> 3
table.insert (t, "d")
print (#t)                          --> 4

t [6] = "8"
-- Temos um valor nil no meio do array (t[5])
print (#t)                          --> ??

t [5] = "e"
-- Agora o array não tem "buracos"
print (#t)                          --> 6
```

If

- Estrutura de controle básica de Lua:

```
local x = 44
if x > 30 then
  print ("maior que 30")
elseif x > 20 then
  print ("maior que 20")
elseif x > 10 then
  print ("maior que 10")
else
  print ("que x pequeno")
end
```

While

- Declare sempre a variável que controla o laço como local

```
local i = 0
while i < 10 do
  print (i)
  i = i + 1
end
```

Laços - *for* numérico

- Usando o for numérico:

```
for var=exp1, exp2, exp3 do
   <corpo>
end
```

- Exemplo:

```
t = {"um", "dois", "três"}

local n = #t

for i=1, n do
   print (t [i])
end
```

Laços - *for* genérico

- Pecorre os valores retornados por uma função iteradora;

```
--imprime os valores do array a
for i, v in ipairs (a) do
   print (v)
end
```

- Lua fornece várias delas:
  - ipairs;
  - pairs;
  - io.lines;

ipairs

- Usado para percorrer arrays:

```
local t = {"um", "dois", "três"}

for i, v in ipairs (t) do
   print (i, v)
end
```

- Percorre até encontrar um valor nil:

```
local t = {"um", "dois", nil, "três"}

for i, v in ipairs (t) do
   print (i, v)
end
```

pairs

- Usado para pecorrer todas as chaves de um tabela:

```
local t = {"um", "dois", nil, "três"}

for k, v in pairs (t) do
   print (k, v)
end
```

- Chaves podem ser numéricas ou não:

```
local t = {one = "um", two = "dois", three = "três"}

for k, v in pairs (t) do
   print (k, v)
end
```

Saindo de um bloco

- Para sair de um bloco, utilize os comandos break ou return
- Um comando break ou return deve ser o último comando de um bloco

```
i = 5
while i < 10 do
  break       -- Erro! break deve ser o último comando
  i = i + 1   -- Atribuição é último comando do "while"
end
```

Saindo de um bloco

- Reorganize o código ou coloque um bloco do-end:

```
i = 5
while i < 10 do
  do
    break    -- Ok! break é o último comando do bloco
  end
  i = i + 1  -- Atribuição é último comando do "while"
end
```

Funções

- Valores de primeira classe;
- Podem ser redefinidas;
- Múltiplos valores de retorno;
- Bom suporte para programação funcional.

Declaração

- A declaração

```
function foo ()
   return 1, 2, 3
end
```

- É açúcar sintático para:

```
foo = function () return 1, 2, 3 end
```

Funções

- Variável foo não possui nenhum significado especial:

```
local a = foo

foo = "bola"

print (a ())    --> 1   2   3
print (foo)     --> bola
```

Parâmetros

- Lua ajusta automaticamente o número de parâmetros de uma função;

```
function foo (a, b)
   print (a, b)
end
```

- Parâmetors não fornecidos recebem nil e parâmetros extra são descartados:

```
foo (4)          --> 4    nil
foo (4, 5)       --> 4    5
foo (4, 5, 6)    --> 4    5
```

Parâmetros

- Indicamos um número variável de parâmetros com $\cdots$
- Exemplo:

```
function add (...)
   local s = 0

   -- percorre a lista de parâmetros
   for i, v in ipairs {...} do
      s = s + v
   end

   return s
end

print (add (3, 4, 10, 25, 12)) --> 54
```

Parâmetros

- Exemplo:

```
function maior3 (...)

  for i, v in ipairs {...} do
    if #v > 3 then   -- # é o operador de tamanho
      print (v)
    end
  end

end

maior3 ("bola", "sol", "lua", "balao")
```

Retornando

- Função pode retornar 0, 1 ou múltiplos valores;
- Nem sempre é possível obter todos os valores retornados por uma função

```
function foo (a, b)
  local x = a or 1
  local y = b or 1
  return x + y, x * y
end
```

Retornando

- Chamando foo:

```
a, b, c, d = foo (1, 2), foo (3, 4), foo (5, 6)

print (a, b, c, d)     -->  3   7   11   30
```

Retornando um único valor

- Coloque parênteses ao redor da chamada para retornar um único valor:

```
a, b = foo (5, 10)
print (a, b)              --> 15   50
a, b = (foo (5, 10))
print (a, b)              --> 15   nil
a, b, c = foo (1, 2), (foo (3, 4))
print (a, b, c)           -->  3   7   nil
```

Valores de Primeira Classe

- Qualquer variável Lua pode armazenar uma função;
- Podemos passar funções como argumentos para outras funções;
- Funções podem retornar funções.

map

- Definição de map:

```
function map (f, t)
  for k, v in pairs (t) do
     t [k] = f (v)
  end
end
```

Usando map

- Incrementando elementos da tabela:

```
function inc (v)
  return v + 1
end
t = {1, 2, 3}
map (inc, t)
```

Usando map

- Definindo a função de incremento durante a chamada:

```
t = {1, 2, 3}
map (function (v) return v + 1 end, t)
```

Fechos (*Closures*)

- Variáveis externas podem ser associadas a funções
- Funções possuem um estado
- Empacotamos a função junto com as variáveis externas, criando um fecho

Fechos

- Exemplo:

```
function generateinc (init, step)
  local n = init or 0
  local s = step or 1
  return function ()
    n = n + s  -- referencia as variáveis n e s
    return n
  end
end
```

Fechos

- Usando generateinc:

```
local i = 3

local inc = generateinc ()
print (inc ())                --> 1

local inc_ = generateinc (i)
print (inc (), inc_ ())       --> 2   4

i = 10
print (inc (), inc_ ())       --> 3   5

local inc10 = generateinc (5, i)
print (inc (), inc_ (), inc10 ())   --> 4   6   15
```

Treinamento

- Conceitos Básicos (funções, tabelas, etc)
- Orientação a Objetos em Lua
- Descrição de dados usando Lua
- Usando a API C de Lua
  - Scripts Lua para configurar aplicações C
  - Exportando funções e objetos C para Lua
- smedeiros@inf.puc-rio.br
- carregal@fabricadigital.com.br

Fim

www.lua.org