Linguagem de scripting

Márcio Lopes Cornélio
CIn-UFPE

Linguagens de Programação

- Tradicionais
  - Aplicações auto-contidas
    - Aceitação uma entrada, manipulação e geração de saída
  - Ênfase em eficiência, manutenção, portabilidade e detecção estática de erros
  - Sistema de tipos
    - Envolve conceitos associados a hardware: números de ponto-flutuante, caracteres, arrays
- Coordenação entre programas

Linguagens de Programação

- Scripting
  - Ênfase em flexibilidade, desenvolvimento rápido, verificação dinâmica (em tempo de execução)
  - Sistema de tipos
    - Envolve conceitos como tabelas, padrões, listas e arquivos
  - *Glue*
    - Combinação de componentes escritos em outras linguagens
  - Extensão

Linguagens de scripting

- Características comuns
  - Modos batch e interativo
- Economia de expressões
  - Evitam declarações extensas
- Falta de declarações, regras de escopo simples
  - Declarações globais são comuns
  - Em certas linguagens (PHP, por exemplo), toda declaração é local (default)
- Tipificação dinâmica flexível
  - Dinamicamente tipadas
    - Verificação antes do uso
    - Contexto determina a interpretação

Linguagem Lua

Caracteríticas

- Extensibilidade
  - Projetada para ser estendida com Lua mesmo ou com outra linguagem
- Simplicidade
  - Simples, pequena. Poucos conceitos, mas poderosos. Fácil de aprender
- Eficiência
  - Implementação eficiente. Uma das mais rápidas linguagens de scripting
- Portabilidade
  - Plataformas: PlayStation, Xbox, Mac OS X, Windows, Unix*

Chunks de código

- Cada peça de código que Lua executa (arquivo ou uma única linha em modo interativo) é chamado de *chunk*
- Não são necessário separadores, mas pode-se (aconselha-se) o uso de ponto-e-vírgula
- Exemplo de *chunks* válidos e equivalentes

```
a = 1
b = a*2

a = 1;
b = a*2;

a = 1 ; b = a*2

a = 1 b = a*2
```

Identificadores

- Lua é *case-sensitive*
- Identificadores
  - Cadeias de caracteres, dígitos e underscore, não pode começar com dígito
- Palavras-reservadas

```
and   break   do   else elseif
end   for   if   in   function
local   nil   not   or
repeat
then   until   while
return
```

Tipos e valores

- Lua é dinamicamente tipada
- Não há definição de tipos na linguagem, cada valor tem seu próprio tipo
- Seis tipos básicos
  - *nil , boolean, number, string, userdata, function, thread* e *table*
- Função *type* retorna o tipo de um valor

```
print(type("Hello world" )
print(type(10*3))
```

Variáveis

- Não possuem valores pré-definidos, qualquer variável contém qualquer tipo

```
print(type(a))
a = 10
print(type(a))
a = print
print(type(a))
```

Nil

- Tipo com um único valor (*nil*) que tem a propriedade de ser diferente de qualquer outro valor
- Para deletar uma variável global, atribuímos *nil* a ela
- É um não-valor, a ausência de valor útil

Boolean

- Dois valores: **true** e **false**
- Sem monopólio
  - Qualquer valor representa uma condição
  - Comandos condicionais tratam **false** e *nil* como falso; qualquer outro valor, como verdadeiro
  - Atenção: zero e strings vazias são tratados como verdadeiros

Numbers

- Representam os números reais (ponto-flutuante de dupla precisão)
- Não há o tipo inteiro
- Exemplos de constantes numéricas

4 4. .4 0.4 4.57e-3 0.3e12 5e+20

Strings

- Sequência de caracteres
- Qualquer dado binário pode ser armazenado em uma string
- São valores imutáveis: uma mudança em um caracter cria uma nova string com a modificação

```
a = "one string"
b = string.gsub(a, "one", "another")
print(a)
print(b)
```

Strings

- São alvo de gerenciamento automático de memória como outros objetos de Lua (tables, functions etc)
  - Não é preciso alocar e desalocar

```
> print ("one line\nnext line\n\"in quotes\" , 'in quotes'")
one line
next line
"in quotes" , 'in quotes'
```

Strings

- Delimitando com colchetes
  - Útil quando uma string contém um programa

```
page = [[<HTML>
    <HEAD>
    <TITLE>An HTML Page</TITLE>
    </HEAD>
    <BODY>
    <A HREF="http://www.tecgraf.puc-rio.br/lua">Lua</A>
    [[a text between double brackets]]
    </BODY>
    </HTML>
    ]]
    write(page)
```

Conversão de tipos

- Conversão automática entre números e strings
  - Comparação 10 == "10" resulta em falso

```
print("10"+1) --> 11
print("10+1") --> 10+1
print("-5.3e-10"*"2") --> -1.06e-09
print("hello"+1) -- ERROR ("hello" cannot be
converted)
print(10 .. 20) --> 1020
```

Strings

- Função *tonumber*

```
l = io.read()
n = tonumber(l)
if n == nil then
 error(l.." is not a valid number")
else
 print(n*2)
end
```

- Função *tostring*

```
print(tostring(10) == "10") --> 1 (true)
print(10.."" == "10") --> 1 (true)
```

Tabela

- Implementa arrays associativos
  - Pode ser indexado não apenas por números, mas também por strings ou qualquer valor da linguagem, exceto **nil**
- Não tem tamanho fixo
- Único mecanismo de estruturação de dados de Lua, mas poderoso
- Usada para representar
  - arrays ordinários, tabelas de símbolos, conjuntos, registros, filas, etc

Tabela

- Não é um valor ou variável, mas um **objeto**
  - Dinamicamente alocado
  - Manipulação de referências
  - Não há necessidade de declarar tabela em Lua
  - Criadas com uma expressão de construtor

```
a = {}
k = "x"
a[k] = 10
a[20] = "great" value="great"
print(a["x"])
k = 20
print(a[k])
a["x"] = a["x"]+1
print(a["x"])
```

Tabela

- É anônima
- Não há relação fixa entre uma variável que mantém a tabela e tabela em si

```
a = {}
a["x"] = 10
b = a
print(b["x"])
b["x"] = 20
print(a["x"]
a = nil
b = nil
```

- Sem referência, pode ser feita coleta de lixo

Tabela

- Para representar registros, Lua usa o nome do campo como índice

```
a.x = 10   --> mesmo que a["x"]=10
print(a.x) --> mesmo que print(a["x"])
print(a.y) --> mesmo que print(a["y"])
```

- Erro comum: confundir *a.x* com *a[x]*. O primeiro representa *a["x"]*; o segundo, a tabela indexada com o valor da variável *x*

```
a = {}; x = "y"
a[x] = 10;
print(a[x]) --> 10
print(a.x) --> nil
print(a.y) --> 10
```

Tabela

- Usando uma tabela como array ou lista

```
a = {}
for i=1,10 do
 a[i] = io.read()
end
```

- Convenção: em Lua arrays começam em 1
- Operador # retorna o último índice

```
print(a[#a])
a[#a]=nil
a[#a+1]=v
```

```
a = {}
for i=1,10 do
 a[#a+1] = io.read()
end
```

Tabela

- Qual é o tamanho deste array?

```
a = {}
a[1000] = 1
```

- Qualquer índice não-inicializado resulta em *nil*
- Flag para término do array
  - Array com elementos *nil*, assume-se qualquer deles é a marca de fim do array

Tabela

- Dúvida sobre tipos dos índices
  - Conversão explícita

```
i = 10; j =“10”; k = “+10”
a = {};
a[i] = “um valor”
a[j] = “outro valor”
a[k] = “ainda outro valor”
print(a[j])
print(a[k])
print(a[tonumber(j)])
print(a[tonumber(k)])
```

Funções

- Valores de primeira classe em Lua
  - Armazenadas em variáveis, passadas como argumentos para outras funções, retornadas como resultados
- Suporte à programação funcional

Expressões em Lua

Operadores

- Operadores aritméticos
  - ‘+’, ‘-’, ‘*’, ‘/’, ‘^’ (exponenciação), ‘%’ (módulo) e ‘-’ (negacão – unário)
- Operadores relacionais

  < > <= >= == ~=

```
a = {}; a.x = 1; a.y = 0
b = {}; b.x = 1; b.y = 0
c = a
-- a==c mas a~=b

"acaí" < ”açaí" < "acorde“
```

Operadores

- Lógicos
  - and or not
  - **false** *e nil* são falsos, todo o restante verdadeiro

```
print(4 and 5) --> 5
print(nil and 13) --> nil
print(false and 13) --> nil
print(4 or 5) --> 4
print(nil or 5) --> 5
```

  - **and** e **or** usam avaliação em curto-circuito
  - x = x or v é equivalente a if x == nil then x = v end
  - (e and a) or b, é equivalente à expressão de C e ? a : b
    - max = ((x > y) and x) or y

Operadores

- Concatenação

```
print(“Hello” .. “World”)
print(0 .. 1)
a = “Hello”
print(a .. “World”)
print (a)
```

Construtores de tabelas

- Expressões que criam ou inicializam tabelas

```
dias = {"segunda", "terça", "quarta", "quinta", "sexta"}
print(dias[4])

a = {x = 10, y = 20}  -- equivalente a
a = {}, a.x=10, a.y =20
```

- Adição e remoção de campos

```
w = {x=0, y=0, label="console"}
x = {math.sin(0), math.sin(1), math.sin(2)}
w[1] = "outro campo"
x.f = w
print(w["x"])
print(w[1])
print(x.f[1])
w.x=nil
```

Tabelas

- Criando e inicializando uma lista encadeada

```
list = nil
for line in io.lines() do
 list = {next=list, value = line}
end
```

- Percorrendo uma lista

```
local l = list
while l do
 print(l.value)
 l = l.next
end
```

Comandos

- Atribuição
  - Múltipla

```
a, b = 10, 2*x
x , y = y, x
a[i], a[j] = a[j], a[i]
```

- Ajuste de número de valores ao número de variáveis

```
a, b, c = 0, 1
print(a,b,c) --> 0 1 nil
a, b = a+1, b+1, b+2
print(a,b) --> 1 2
a, b, c = 0
print(a,b,c) --> 0 nil nil
```

Comandos

Variáveis locais

- Lua suporta variáveis locais, com o comando **local**

```
j  = 10
local i =1
```

- Escopo limitado ao bloco (estrutura de controle

```
x = 10
local i = 1 -- local ao chunk
while i<=x do
      local x = i*2 -- local ao corpo do while
      print(x)
      i = i+1
end
if i > 20 then
      local x -- local ao corpo do "then"
      x = 20
      print(x+2)
else
      print(x) --> 10 (global)
end
print(x) --> 10 (global)
```

Variáveis locais

- No modo interativo, usar **do**-**end** para delimitar blocos

```
do
      local a2 = 2*a
      local d = sqrt(b^2-4*a*c)
      x1 = (-b+d)/a2
      x2 = (-b-d)/a2
end
print(x1, x2)
```

Variáveis locais

- Declaração sem valor iniciação explícito, é inicializada com *nil*

```
local a, b = 1, 10
if a<b then
      print(a) --> 1
      local a -- '= nil' está implícito
      print(a) --> nil
end -- finaliza o  bloco que começou no 'then'
print(a,b) --> 1 10
```

Estruturas de controle

- if then else

```
if  a < 0 then a = 0 end

if a < b then return a else return b end
```

- Aninhamento

```
if op == "+" then
      r = a+b
elseif op == "-" then
      r = a-b
elseif op == "*" then
      r = a*b
elseif op == "/" then
      r = a/b
else
 error ("invalid operation")
end
```

Estruturas de controle

- while

```
local i = 1
while a[i] do
      print(a[i])
      i = i+1
end
```

- repeat until

```
repeat
      line = os.read()
until line ~= ""
```

For

- Numérico
  - A variável de controle de loop é local e automaticamente declarada
  - Todas as expressões são avaliadas de uma vez, antes do loop iniciar

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

```
for i=10,1,-1 do print(i) end
a = {'a', 'b', 'c', 'd'}

for i=1,f(x) do print(i) end
```

For

- Genérico

```
for i, v in pairs(a) do print(v) end – imprime os valores de a

t = {x = 1, y = 10, z = -4}
for k,v in t do print(k,v) end


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

Funções

Funções

- Computam e retornam valores

```
print(8*9, 9/8)
a = sin(3)+cos(10)
```

- Definição

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

Funções

- Número de argumentos

```
function f(a, b)  return a or b end
```

CHAMADA	PARAMETROS
f(3)	a=3, b=nil
f(3, 4)	a=3, b=4
f(3, 4, 5)	a=3, b=4 (5 é descartado)

```
function inc_count (n)
 n = n or 1
 count = count + n
end

inc_count(5)
inc_count(1) ⇔ inc_count()
```

Funções

- Múltiplos resultados

```
function foo0 () end -- sem resultados
function foo1 () return 'a' end – retorna 1 resultado
function foo2 () return 'a','b' end -- retorna 2 resultados
foo0()
foo1() -- 'a' descartado
foo2() -- 'a' e 'b' descartados
print(foo0()) -->
print(foo1()) --> a
print(foo2()) --> a b
print(foo2()..."x") --> ax
x,y=foo0() -- x=nil, y=nil
x,y=foo1() -- x='a', y=nil
x,y=foo2() -- x='a', y='b'
x=foo2()  -- x='a', 'b' is discarded
```

Funções

- Número de parâmetros variáveis
  - Uma mesma função recebe um número variável de argumentos
  - Argumentos coletados internamente (*varargs*)

```
function add(...)
  local s = 0
  for i,v in ipairs{...} do
    s = s + v
  end
 return s
end
print (add(3, 4, 1, 25))
```

Funções

- Funções como valores de primeira classe
  - Valor como, por exemplo, números e strings
  - Podem ser armazenadas em variáveis (globais e locais) e tabelas
- Escopo léxico
  - Uma função pode acessar variáveis da função em que está inserida
  - Lambda calculus

Funções

- Funções são anônimas em Lua
- `print` é uma variável que mantém uma função

```
a = {p = print}
a.p("Oi")
print = math.sin
a.p(print(1))
sin = a.p
sin(10, 20)
```

Funções como valores

- Declaração de função é um comando (uma atribuição)
- `function (x) .... end` como construtor de funções
- Resultado: função anônima

```
function foo (x) return 2 * x  end

foo = function (x) return 2*x end
```

Upvalue

- Função inserida em outra função tem acesso às variáveis locais da última
  - Escopo léxico
    - A função anônima refere-se a uma variável não-local `i` (*upvalue*) para manter o contador

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

Closure

- Função e acesso a variáveis não-locais
- Cada chamada a função newCounter cria uma nova variável local i

```
c1  = newCounter()
print(c1())
print(c1())

c2  = newCounter()
print(c2())
print(c1())
print(c2())
```

- O *closure* é o valor em Lua, não a função

Redefinições de funções

- Por serem armazenadas em variáveis, funções podem ser redefinidas
- Exemplo: usando a definição original na nova implementação

```
oldSin = math.sin
math.sin = function (x)
 return oldSin(x*math.pi/180)
end
```

```
do
  local oldSin = math.sin
  local k = math.pi/180
  math.sin = function (x)
   return oldSin(x*k)
  end
end
```

Funções não-globais

- Função local (restrita a um escopo)
- Função local visível em um *chunk*

```
local f = function (...)
 ...
end
local g = function (...)
  ...
  f() – 'f' externo visível
  ...
end
```

Funções não-globais

- Situação em que a função local ainda não está definida

```
local fact = function (n)
   if n == 0 then return 1
  else return n*fact(n-1) -- buggy
  end
end
```

- Corrigindo

```
local fact
 fact = function (n)
  if n == 0 then return 1
  else return n*fact(n-1)
 end
end
```

Iterador e *closures*

- Construção que permite iterar sobre elementos de uma coleção
- Todo iterador mantém algum estado entre chamadas sucessivas
  - Solução: *closures*

```
function value (t)
  local i = 0
  return function () i = i + 1; return t[i] end
end
```

Iterador e *closures*

- Iterador para lista
- A função *values* é uma fábrica. Cada chamada cria um novo *closure* que mantém no estado as variáveis t e i

```
function values (t)
  local i = 0
  return function () i = i + 1; return t[i] end
end
```

Corrotinas

Corrotinas

- Similar a thread
  - Linha de execução com pilha, variáveis locais e apontador de instruções
  - Compartilha variáveis globais
- São colaborativas
  - Em dado momento, um programa com corrotinas executa apenas uma delas
  - A corrotina em execução suspende a própria execução apenas quando explicitamente requisita suspensão

Corrotinas

- Tabela `coroutine`
- Funções: `create`, `status`, `resume`, `yield`
- Argumento para create frequentemente é uma função anônima

```
co = coroutine.create(function () print("oi") end)
print(co)
```

Estados

- Suspensa, rodando, morta e normal
- Após criada, o estado é *suspended*

```
print(coroutine.status(co))
```

- Estados: mudando de *suspended* para *running*
  - Função *resume*

```
print(coroutine.resume(co))
```

- Após executar o corpo da corrotina muda o estado para *dead*

Função *yield*

- Permite que uma corrotina rode até que suspenda a própria execução para ser retomada depois

```
co = coroutine.create(function ()
        for i=1,10 do
          print("co", i)
          coroutine.yield()
        end
      end)
print(coroutine.resume(co))
```

- Quando uma corrotina retoma uma outra, a primeira não fica suspensa (ou não poderia ser retomada), vai para o estado *normal*

Troca de dados

- Par *resume-yield*
- O primeiro resume sem yield correspondente, passa os argumentos extras para a função principal da corrotina

```
co = coroutine.create(function (a,b,c)
       print("co", a,b,c)
     end)

coroutine.resume(co, 1, 2, 3)
```

Troca de dados

- Uma chamada a *resume* (não havendo erros) retorna quaisquer argumentos passado para o *yield* correspondente

```
co = coroutine.create(function (a,b)
       coroutine.yield(a + b, a - b)
     end)

print(coroutine.resume(co, 20, 10)) --> true 30 10
```

Troca de dados

- Uma chamada a *yield* retorna quaisquer argumentos extras passada para o *resume* correspondente

```
co = coroutine.create (function ()
       print("co", coroutine.yield())
     end)


coroutine.resume(co)
coroutine.resume(co, 4, 5) --> co 4 5
```

Troca de dados

- Quando uma corrotina finaliza, quaisquer valores passados para a função principal vão para o *resume* correspondente

```
co = coroutine.create(function ()
       return 6, 7
     end)


print(coroutine.resume(co)) --> true 6 7
```

Características

- Corrotinas assimétricas
  - Funções distintas para suspender a execução e para retomá-la
  - Obs: Seria simétrica se houvesse apenas uma função para suspender e retomar a execução
- Semi-corrotina
  - Corrotina pode suspender a execução apenas quando não está chamando outra função (não chamadas pendentes na pilha de controle)
    - Apenas o corpo do main pode fazer *yield*

Produtor-consumidor

```
function producer ()
  while true do local x = io.read() -- produz novo valor
    send(x) -- envia para consumidor
end

function consumer ()
   while true do local x = receive() -- recebe do produtor
     io.write(x, "\n") -- consome novo valor
   end
end
```

- Executam sempre
- Problema
  - Como casar *send* e *receive*?

Produtor-consumidor

- Usando o par *resume-yield*
  - Uma chamada a *yield*, não entra em uma nova função, mas retorna para uma chamada pendente (para retomar execução)
  - Uma chamada a *resume* não inicia uma nova função, mas retorna uma chamada a *yield*

Produtor-consumidor

- Combinar um *send* com um *receive*, de modo que ajam com mestre e escravo
  - *receive* retoma (*resume*) a execução do produtor
  - *send* envia (*yield*) o novo valor de volta para o consumidor

```
function receive ()
  local status, value = coroutine.resume(producer)
  return value
end

function send (x)
   coroutine.yield(x)
end
```

Produtor-consumidor

- Produtor como corrotina

```
producer = coroutine.create(
  function ()
   while true do
   local x = io.read() -- produz novo valor
   send(x)
  end
end)
```

- Programa começa chamando um consumidor
  - A cada item necessário, o consumidor retoma o produtor, este executa até ter dado para o consumidor, então para até o consumidor reiniciar a execução do produtor

Filtros

- Tarefa entre o produtor e o consumidor
  - Realiza transformação nos dados
- Consumidor e produtor ao mesmo tempo