LEONARDO SANTOS PAULUCIO
LUIZ GUILHERME LITTIG BERGER

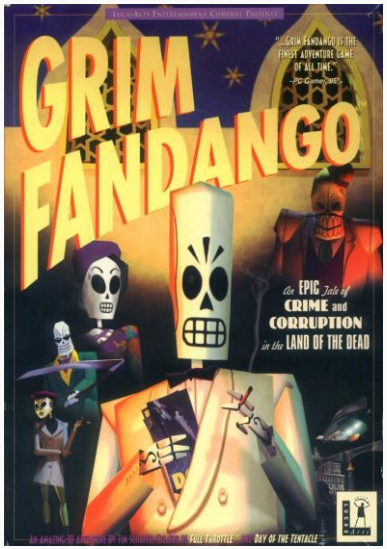
Sumário

- História
- Características
- Escopo
- Operadores Lógicos e Aritméticos
- Estruturas de Controle
- Funções
- Tables / *Metatables* / *Metamethods* / *OO*
- Modularização
- Polimorfismo
- Concorrência
- Exceções
- Avaliação da Linguagem


Lua
Lua
Made in Brazil

História

- Foi projetada, implementada e desenvolvida no Brasil, por uma equipe na PUC-Rio em 1993.
- Em 1997, foi usada pela primeira vez como linguagem script pela *LucasArts* no jogo *Grim Fandango.*
- Semelhanças com algumas linguagens:
  - Icon, concepção;
  - Python, facilidade de aprendizado;
  - Lisp e Schema, estrutura dos dados.



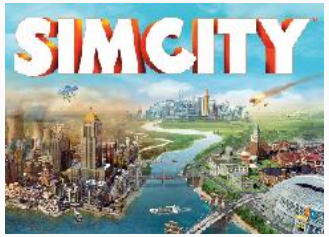
Utilização

- Hoje é usada em:
  - ➢ Robótica;
  - ➢ Processamento de imagens;
  - ➢ Bio-informática;
  - ➢ Aplicações industriais

    Ex: Adobe's Photoshop Lightroom;
    - ➢ Em sistemas embutidos

    Ex: Middleware Ginga para TV digital;
    - ➢ Jogos

    Ex: World of Warcraft, Angry Birds e SimCity.



Como obter?

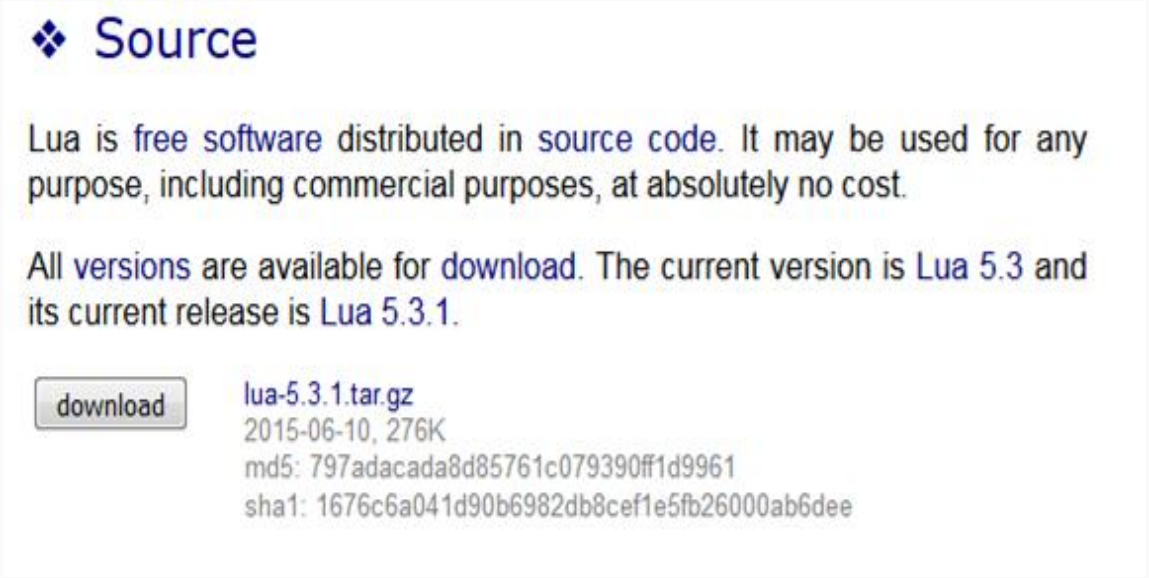
Código fonte, executáveis, ferramentas.

Documentação técnica, manual, *papers* e livros

Suporte, *wiki*, grupos.

www.lua.org

Linux



Source

Lua is free software distributed in source code. It may be used for any purpose, including commercial purposes, at absolutely no cost.

All versions are available for download. The current version is Lua 5.3 and its current release is Lua 5.3.1.

download

lua-5.3.1.tar.gz
2015-06-10, 276K
md5: 797adacada8d85761c079390ff1d9961
sha1: 1676c6a041d90b6982db8cef1e5fb26000ab6dee

Ou simplesmente digite no terminal:

sudo apt-get install lua5.2

Windows / MAC

Download

All the binaries, source code and documentation are available from the SourceForge proje

http://sourceforge.net/projects/luabinaries/files/

But here are shortcuts for the most popular downloads:

LuaBinaries 5.3 - Release 1

lua-5.3_Sources.tar.gz	Source Code, Makefiles and IDE Projects
lua-5.3_Sources.zip	Source Code, Makefiles and IDE Projects
lua-5.3_Win32_bin.zip	Windows x86 Executables
lua-5.3_Win32_dllw4_lib.zip	Windows x86 DLL and Includes (MingW 4 Compatible)
lua-5.3_Win64_bin.zip	Windows x64 Executables
lua-5.3_Win64_dllw4_lib.zip	Windows x64 DLL and Includes (MingW 4 Compatible)
lua-5.3_MacOS109_bin.tar.gz	MacOS X Intel Executables
lua-5.3_MacOS109_lib.tar.gz	MacOS X Intel Library and Includes

http://luabinaries.sourceforge.net/download.html

Características

- Linguagem multi-paradigma;
- Sintaxe simples e intuitiva;
- Dinamicamente tipada;
- Linguagem extensível;
- *Case-sensitive*;
- Linguagem ideal para *scripting* e prototipagem rápida;
- Linguagem interativa;
- É uma linguagem de código aberto.

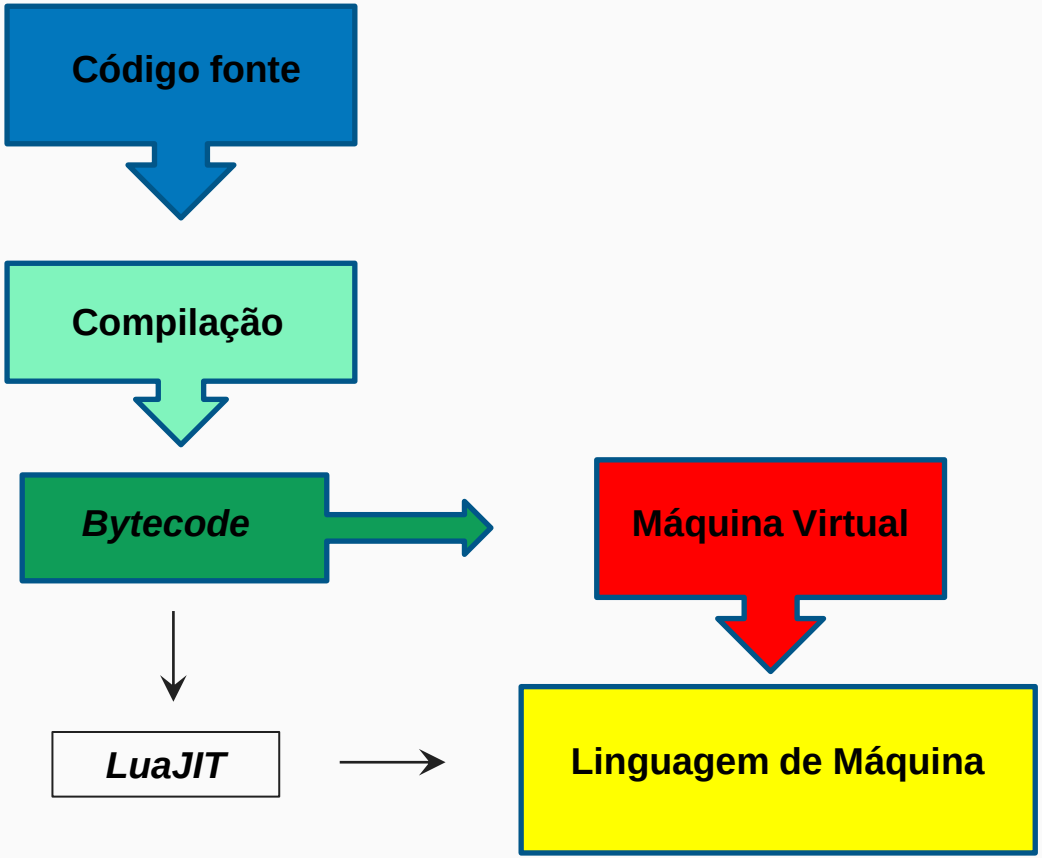
Execução

- É interpretada a partir de *bytecodes*(*luac.out*) para uma máquina virtual baseada em registradores;

- Tem gerenciamento automático de memória com coleta de lixo incremental;

- Para se obter maior velocidade pode se usar o *LuaJIT*, que é um compilador just-in-time.

Execução


Código fonte
Compilação
Bytecode
Máquina Virtual
LuaJIT
Linguagem de Máquina

Tipos de dados

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

Objetos

Em Lua, não existe definição de tipo, cada valor carrega seu próprio tipo.

Todos os valores são de **primeira classe**.

A função type() retorna o tipo de um valor qualquer:

```
print(type(true))        --> boolean
```

Nil

- Tipo de valor único, **nil**;
- Variáveis, por padrão, tem o valor nil, até que sejam referenciadas;
- Tem a única finalidade de apresentar se uma variável que não é útil(não possui valor);
- Nil pode ser atribuído a uma variável para deletá-la.

Boolean

- Todo valor tem uma condição a ele associado;
- Apenas **false** e **nil** são considerados falsos, e todo o resto verdadeiro.

```
3 if 0
4     then
5         print("Verdadeiro");
6     else
7         print("Falso");
8 end
```

Number

- Ponto flutuante de dupla precisão;
- Podemos compilar Lua para que trabalhe com outros tipos de números, como longs.

```
> a = 1e14
> print(a)
1e+14
> a = 99999999999999
> print(a)
99999999999999
```

NaN, valor numérico especial usado para representar valores indefinidos ou não representáveis.
Ex: 0/0

Strings

- Possuem valores imutáveis;
- Manipulação através de funções de bibliotecas.


```
1 a = "uma string" -- 'uma string'
2 b = string.gsub(a, "uma", "outra")
3
4 print(a)        --> uma string
5 print(b)        --> outra string
```

Strings

- Delimitadores
  - Aspas Duplas; Ex: a = “Isto é um teste”
  - Aspas Simples; Ex: b = ‘Isto é outro “teste” ’
  - Duplos Colchetes; Ex c = [[Essa é uma string em duas linhas]]

- O caractere \ indica sequências de escape;
  - \n, \t, \r, \’, \”, \\

Tables

- Arrays associativas;
- São considerados objetos.
- Manipulam referências;
- Única forma de estruturação de dados de Lua;
- Podem ser heterogêneas.

```lua
a = {}
k = "x"
a[k] = 10
print(a["x"])     --> 10
a["x"] = a["x"] + 1  --  a.x = a.x + 1
print(a[k])     --> 11
```

Function

- Uma função pode ser passada como parâmetro de uma função, armazenada em uma variável ou retornada como resultado (**primeira classe)**;

- Lua apresenta uma forte aplicação para linguagem funcional.

Userdata

- Permite o armazenamento de dados em C em variáveis Lua;
- Lua não possui operações pré-definidas para manipular o tipo *userdata,* devendo ser feita através de um API de C.
- Existem dois tipos:
  - *Userdata Completo*, bloco de memória gerenciado por Lua.
  - *Userdata Leve*, bloco de memória gerenciado pelo "hospedeiro"

Threads

- As threads podem ser implementadas através de ***coroutines***;
- Não são relacionadas com as *threads* dos sistemas operacionais;
- Lua suporta *coroutines* em todos sistemas, até aqueles que não suportam *threads* nativamente.

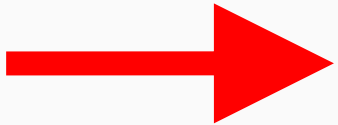
Convenções Léxicas da Liguagem

- Identificadores em Lua podem ser qualquer *String* de letras, dígitos e *underline;*
- Mas não podem começar com dígitos numéricos;
- Deve se evitar usar identificadores com *underline* seguido de letras maiúsculas, pois elas são reservadas para usos especiais da linguagem.
- É *case sensitive*.

Convenções Léxicas da Liguagem

- Comentários de linha são iniciados por -- (dois hífens);
- Comentários de bloco são iniciados --[[e fechados com --]]

```
1 print("Ola")
2 --print("Estou comentado")
3 -- s = "Hoje é quarta"
4 print(s)
5 --[[
6 h = 2015
7 print(h)
8 --]]
9 print("Adeus")
```



```
Ola
nil
Adeus
```

Convenções Léxicas da Liguagem

- Apresenta as seguintes palavras reservadas:

and	break	do	else	elseif
end	false	for	function	if
in	local	nil	not	or
repeat	return	then	true	until
while	goto	module	require	

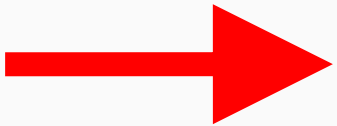
Escopo

- Variáveis Globais;
- Variáveis Locais;
- Upvalues.

Variáveis Globais

- Por *default* todas as declarações de variáveis são assumidas serem globais.

```
1 raio = 2
2
3 function comprimento()
4         return raio*2*3.14
5 end
6
7 print(comprimento())
8 print(s)
```



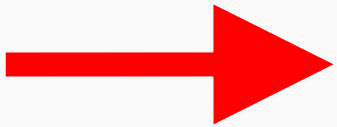
```
12.56
nil
```

Variáveis Locais

- Podem ser declaradas em qualquer lugar dentro de um bloco de comandos;
- Necessário usar a palavra “local” antes da variável;
- Têm escopo somente dentro do bloco;
- Quando declaradas com o mesmo nome de uma variável global, encobrem o acesso à variável global naquele bloco;
- Em Lua o acesso à variáveis locais é mais rápido do que à variáveis globais.

Variáveis Locais

```lua
raio = 2
print(raio)

function comprimento()
        local raio = 10
        print("Raio = ",raio)
        return raio*2*3.14
end

print(raio)
print(comprimento())
```



```
2
2
Raio =  10
62.8
```

Variáveis Locais

- É possível criar um bloco com **do… end**;

```
1  x = 10                  -- variável global
2      do                  -- novo bloco
3        local x = x       -- novo 'x', com valor 10
4        print(x)          --> 10
5        x = x+1
6        do                -- outro bloco
7          local x = x+1   -- outro 'x'
8          print(x)        --> 12
9        end
10       print(x)          --> 11
11     end
12     print(x)            --> 10  (o x global)
```

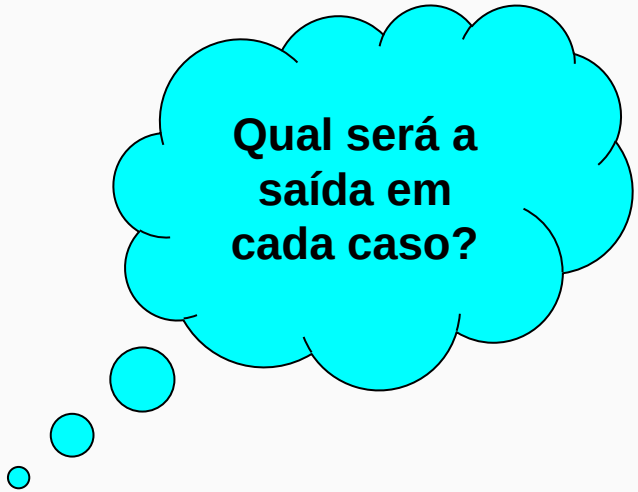
```
10
12
11
10
```

Variáveis Locais

- É recomendável sempre declarar variáveis dentro de funções como locais;


```
function f1()
    local a = 1
end
```

```
function f1()
    a = 1
end
```




```
f1()
print(a)
```

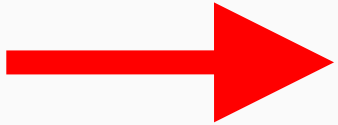
Upvalues

- Variáveis locais podem ser acessadas por funções definidas dentro de seu escopo.
- Uma variável local usada por uma função mais interna é chamada de ***upvalue***, ou de variável local externa, dentro da função mais interna.
- Cada execução de um comando local define novas variáveis locais.

Upvalues

- Exemplo:

```
1 a = {}
2 local x = 20
3 for i=1,4 do
4         local y = i
5         a[i] = function () y=y+1; return x+y end
6         print(a[i]())
7 end
```



```
22
23
24
25
```

Atribuição

- Atribuição feita pelo simbolo operador = (igual);
- Lua permite atribuição simples ou múltipla;
- Pode se utilizar ponto-e-vírgula(;) ou não ao final de um comando;

Simples:

```
1 a = 2;
2 b = "Ola";
```

Múltipla:

```
1 g,h = “Teste”, 3;
2 b, h = “Ola”;   -- h recebe nil
3 j, t = “True”, 1, false; -- false é descartado
4 a,b = b,a  --Troca os valores entre a e b
```

Operadores Aritméticos

- Lua oferece os operadores aritméticos padrões;

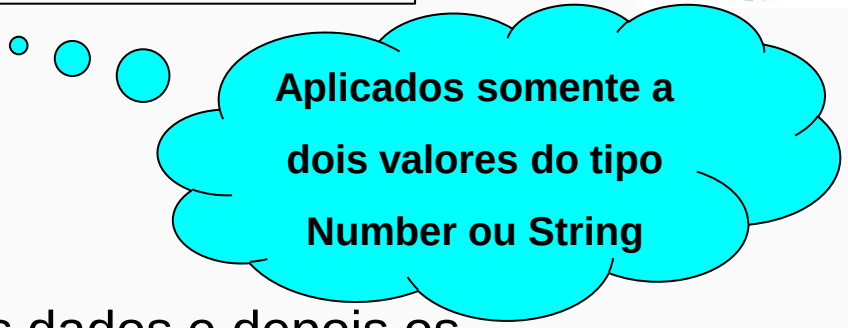
- Só podem ser aplicados a valores do tipo *Number* ou *String* que possam ser convertidos para *Number*.

+ - / * % - () ^

Operadores Relacionais

- Operadores padrões:

~= == < > <= >=



- A igualdade compara primeiro o tipo dos dados e depois os valores;
- Tables, threads e userdatas são comparados por referência.

Operadores Lógicos

- São operadores lógicos em Lua:

  and **or** **not**

- **and** retorna o primeiro argumento se ele for *false* caso contrário retorna o segundo;
- **or** retorna o primeiro argumento se ele for diferente de *false*, caso contrário retorna o segundo;
- Apresentam avaliação de curto-circuito;
- Os operadores lógicos bit a bit: &(AND), |(OR), ^(XOR) e (NOT)foram adicionados na versão 5.3.

Operadores Lógicos

- Exemplo:


```
1 a = 1 and "ola"          -->    ola
2 b = nil and 1            -->    nil
3 c = 1 or false           -->    1
4 d = false or 10          -->    10
5 e = 2 <= 10              -->    true
6 f = 3 >= 20              -->    false
7 g = "Ola" == "Ola"       -->    true
8 h = "quarta" < "asa"     -->    false
9 i = true ~= h            -->    true
```

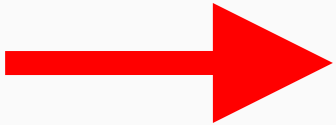
Outros Operadores

- Operador de Concatenação
  - Esse operador é representado por dois caracteres ponto (..);
  - Aplicável a valores do tipo String;
  - Convertem valores do tipo Number quando concatenados a String.

Outros Operadores

- Exemplo:

```lua
a = "Linguagem"
b = "Lua"
c = 3.2
c = a .. " " .. b .. c
print(c)
```

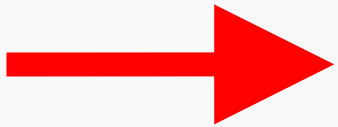


```
Linguagem Lua3.2
```

Outros Operadores

- Operador de Comprimento
  - Denotado pelo operador unário #;

```lua
a = {}
a[1] = "Linguagem"
a[2] = "de"
a[3] = "Programação"

print(a)

for v, k in pairs (a) do
print(k)
end

print(#a)
```



```
table: 0x1b54ea0
Linguagem
de
Programação
3
```

Precedência

- A precedência de operadores em Lua é a seguinte:

Maior Precedência

```
or
and
<    >    <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unário)
^
```

➔ Podem ser usados parênteses para mudar as precedências de uma expressão.

➔ Os operadores de concatenação ('..') e de exponenciação ('^') são associativos à direita.

➔ Todos os outros operadores binários são associativos à esquerda.

Estrutura de Controle

- Lua possui as seguintes estruturas de controle:
  - If-else;
  - While;
  - Repeat-until;
  - For:
    - númerico;
    - genérico;

If-else

- Sintaxe:


```
if expr then
   bloco
end
```

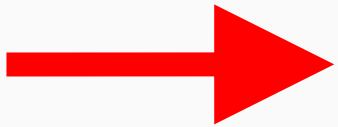
```
if expr then
   bloco1
else
   bloco2
end
```

```
if expr1 then
   bloco1
elseif expr2
   bloco2 …
elseif exprN
   blocoN
else
   blocoN+1
end
```

If-else

- Exemplo

```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
```



```
maior que 30
```

While

- O comando *while* apresenta a seguinte sintaxe:

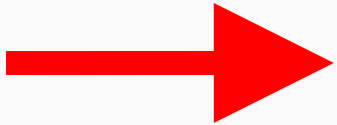
```
while expr1 do
    bloco
end
```

- Onde *expr1* é a condição que é verificada em cada *loop*;

- E *bloco* são os comandos que serão realizados em cada *loop*.

While

```
 7 while i < 10 do
 8     print (i)
 9     i = i + 1
10 end
```



```
0
1
2
3
4
5
6
7
8
9
```

For Numérico

- Sintaxe:

```
for var = exp1, exp2, exp3 do
    corpo_do_laço
end
```

- Onde *exp1* e o valor inicial de *var*;
- O corpo do laço será executado enquanto *var* for menor ou igual *exp2*, no caso de um passo positivo, ou maior ou igual a *exp2*, no caso de um passo negativo.
- A expressão *exp3* representa o passo do laço, quando não especificado é 1.
- Todas as três expressões de controle são avaliadas somente uma vez, antes do laço começar. Elas devem todas resultar em números;
- A variável var é local, só existe dentro do laço.

For Numérico

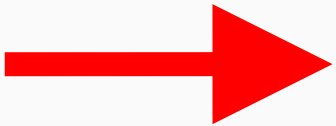

```lua
print("Passo implicito")

for i=1, 3 do --passo implicito = 1, enquanto var <= 3
    print (i)
end

print("Passo explicito")

for i=3, 1, -1 do -- passo explicito = -1, enquanto var >= 1
    print (i)
end
```

```
Passo implicito
1
2
3
Passo explicito
3
2
1
```

For Numérico

- Quando não estamos interessados em uma variável é comum utilizar _ ;

```lua
t = {1,nil,false,4.98,"segunda"}
for _, v in pairs (t) do
    print (v)
end
```



```
1
false
4.98
segunda
```

For Genérico

- Usado para percorrer os valores retornados por uma função iteradora.

- Os principais iteradores fornecidos por Lua são: *pairs*, para percorrer as chaves de uma tabela; *ipairs*, para percorrer os índices de um *array*; e *io.lines*, para percorrer as linhas de um arquivo.

For Genérico

- Sintaxe

```
for var_1, ... , var_n in listaexps do
    bloco
end
```

- Onde *listaexps* é a função iteradora;

For Genérico

- Imprimindo todos os seus valores de um *array*:

```
for i, v in ipairs (a) do -- i guarda o índice, v o valor
    print (v)
end
```

- Imprimindo todas as chaves de uma tabela t:

```
for k, v in pairs (t) do -- k guarda a chave, v o valor
    print (k, v)
end
```

Repeat-until

- Semelhante ao *do-while;*
- Sintaxe:

```
repeat
    bloco
until expr
```

- Exemplo:

```
i = 4
repeat
    i = i - 1
    print(i)
until (i == 0)
```

```
3
2
1
0
```

Estruturas de controle

- Possui os comandos *break, goto* e *return*;


```lua
a = {1,2,3,4}

for v,i in ipairs (a) do
    if v == 3 then
        break
    else
        print(v)
    end
end
```

```lua
for x = 1 ,10 do
    print(x)
    if x == 4 then
        goto skip
    end
end
:: skip ::
```

```lua
function fatorial(n)
    local x = 1
    for i = 2, n do
        x = x * i
    end
    return x
end
```

Funções

- Uma definição de função é uma expressão executável, cujo valor tem tipo *function*.

- Quando Lua pré-compila um trecho, todos os corpos de funções do trecho são pré-compilados também.

- Sempre que Lua executa a definição de uma função, a função é instanciada (ou fechada). Essa instância de função (ou fecho) é o valor final da expressão;

Funções

- São cidadãos de primeira classe;
- Pode possuir desde nenhum até centenas de parâmetros;
- Pode não retornar nenhum valor ou vários;
- Valores do tipo *string* e *number* são passados por valor;
- Valores do tipo *table, function* e *userdata* são passados por referência;
- Passagem de parâmetros é posicional.

Funções

- Existem as seguintes sintaxes para se declarar uma função

```
function nomedafuncao(arg_1,..., arg_n)
    corpoDaFuncao
end
```

```
nomedafuncao = function(arg_1,...,arg_n)
    corpoDaFuncao
end
```

Funções


```lua
function foo (a, b)
    local x = a or 1 -- x recebe o valor padrao 1 quando a e um valor falso
    local y = b or 1 -- y recebe o valor padrao 1 quando b e um valor falso
    return x + y, x * y
end
```

```lua
foo = function(a,b)
        local x = a or 1  -- x recebe valor 1 quando a é falso
        local y = b or 1  -- y recebe valor 1 quando b é falso
        return x + y, x * y
end
```

Funções

- É possível chamar uma função com um número diferente de parâmetros do declarado na função.

- Caso seja passado mais parâmetros do que a função espera os valores extra são descartados.

- Caso seja chamada com um número menor de parâmetros, os valores dos parâmetros que não foram fornecidos recebem nil.

Funções


```
1 s, p = foo (3, 4) -- a e 3 e b e 4
2 print (s, p) --> 7 12
3 s, p = foo (2) -- b e nil e y e inicializado com 1
4 print (s, p) --> 3 2
5 s, p = foo (2, 5, 8) -- 8 e descartado
6 print (s, p) --> 7 10
7 print (foo ()) --> 2 1
```

Funções

- Lua fornece mecanismos para funções com número variáveis de parâmetros;

- Essas podem ser declaradas usando-se três pontos (...);

- Para acessar os parâmetros chamados, usa-se a notação {...}, para criar um *array*;

Funções


```lua
function maior3 (...)
    for i, v in ipairs {...} do -- percorre a lista de parametros
        if #v > 3 then -- # e o operador de tamanho para cadeias e arrays
            print (v)
        end
    end
end

maior3 ("bola", "sol", "lua", "balao") -- Imprime "bola" e "balao"
```

Funções

- É possível retornar somente o primeiro valor de uma função;

- Coloca-se parênteses ao redor da chamada da função.

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

Closures

- Lua permite a criação de *closures*(fechos);
- Associam variáveis externas com uma função que vai ser definida;


```lua
local i = 3

function generateinc (init, step)
    local n = init or 0
    local s = step or 1

    return function ()
                n = n + s -- referencia as variaveis externas n e s
                return n
            end
end
```

Closures


```
13 local inc = generateinc ()
14 print (inc ()) --> 1
15
16 local inc_ = generateinc (i)
17 print (inc (), inc_ ()) --> 2 4
18
19 i = 10
20 print (inc (), inc_ ()) --> 3 5
21
22 local inc10 = generateinc (5, i)
23 print (inc (), inc_ (), inc10 ()) --> 4 6 15
```

Tabelas

- Único meio de estruturação de dados;
- Arrays associativos;
- Não possuem tamanho fixo;
- Poderoso para implementação de vetores, matrizes, filas, pilhas, etc;

Tabelas


```
 2 tabela = {}
 3 tabela = {x = 5}
 4 print ( tabela ["x"]) --> 5
 5 print ( tabela.x ) --> 5
 6
 7 a = {}
 8 x = "y"
 9 a[x] = 10 -- atribui 10 ao campo "y"
10 print (a[x]) --> 10 -- valor do campo "y"
11 print (a.x) --> nil -- valor do campo "x"
12 print (a.y) --> 10 -- valor do campo "y"
```

Tabelas

- Pode-se utilizar a biblioteca padrão de Lua *table,* para realizar manipulações;
- Contudo, não é necessário utilizá-la;
- Índice 1 representa a posição do primeiro elemento da tabela;

Tabelas

- A função *insert* insere-se um elemento na tabela;

- Caso não receba o índice no qual será inserido, adiciona o elemento na posição n+1 onde n é o tamanho da tabela.

```
2 local t = {}
3 for i=1, 3 do
4     table.insert (t, i)
5 end
```

```
7 print (t[1]) --> 1
8 print (t[2]) --> 2
9 print (t[3]) --> 3
```

Tabelas

- Para se inserir um elemento em uma posição da tabela, é necessário passar o índice para a função

- Nesse caso, os elementos da tabela nas posições 2 e 3 vão ser deslocados:

```
2 table.insert (t, 2, 4)
3 print (t[1]) --> 1
4 print (t[2]) --> 4
5 print (t[3]) --> 2
6 print (t[4]) --> 3
```

Tabelas

- Pode se fazer inserção de modo direto também;


```
10 local t = {}
11 for i=1, 3 do
12     t [i] = i
13 end
```

```
15 local t = {}
16 for i=1, 3 do
17     t [#t++1] = i
18 end
```

Tabelas

- A função *remove* retira um elemento da tabela;
- Caso não receba o índice irá remover o último elemento;


```
3 local t = {10, 20, 30}
4 table.remove (t) -- remove o ultimo elemento
5 print (t[1], t[2], t[3]) --> 10 20 nil
6
7 table.insert (t, 40) -- insere elemento no fim
8 table.remove (t, 2) -- remove o segundo elemento, deslocando os elementos seguintes
9 print (t[1], t[2], t[3]) --> 10 40 nil
```

Tabelas

- É possível remover o elemento diretamente sem utilizar as funções da biblioteca;


```
2 local t = {10, 20, 30}
3 t[3] = nil -- remove o ultimo elemento
4 print (t[1], t[2], t[3]) --> 10 20 nil
5
6 t [#t+1] = 40 -- insere elemento no fim
7 t [2] = nil -- remove o segundo elemento, não desloca os elementos restantes
8 print (t[1], t[2], t[3]) --> 10 nil 40
```

Metatables

- Cada tabela pode possuir uma *metatable* associada;
- Permitem modificar o comportamento de um tabela indicando qual ação deve ser tomada quando uma operação envolvendo uma tabela é realizada.
- Inicialmente, uma tabela não possui uma *metatable associada;*
- Para saber se uma tabela possui uma *metatable* associada podemos chamar a função *getmetatable*:

Metatables

- Podemos assim, por exemplo, implementar a operação de soma entre duas tabelas.
- *Tables* e *userdata* tem *metatables* individuais;
- Os outros tipos possuem apenas uma *metatable* para todos os seus valores;
- Lua permite somente a manipulação de *metatables* de tabelas.

Metatables


```
3 local meta = {}
4 local t = {}
5 setmetatable (t, meta)
6 print (meta, getmetatable (t)) --> table: 0x806f118 table: 0x806f118
```

- Temos agora que *meta* é a *metatable* de t.

Metamethods

- É possível fazer algumas operações envolvendo tabelas, tais como +, - e <, cujo significado é dado através de *metatables*.

```
2 local t1, t2 = {}, {}
3 t1.x = 20 t1.y = 30
4 t2.x = 10 t2.y = 5
5 t1 = t1 + t2     --> ERRO!!!
```

- As tabelas t1 e t2 não possuem uma *metatable* associada indicando como realizar a operação +.
- Lua tenta realizar a operação de soma usual, a qual irá falhar.

Metamethods

- Para se definir a operação + deve-se criar uma *metatable* e definir o seu campo *add*;
- Todas as operações devem iniciar com 2 *underlines* (__) antes do nome;

```lua
local t1, t2 = {}, {}
local meta = {}
setmetatable (t1, meta)
setmetatable (t1, meta)
```

```lua
meta.__add = function (a, b)

    local c = {}
    setmetatable (c, getmetatable (a))
    c.x = a.x + b.x
    c.y = a.y + b.y
    return c
end
```

Metamethods

- Lua primeiramente verifica se a tabela possui uma *metatable* associada;
- Depois, verifica se possui o *metamethod* definido;
- Agora, com a *metatable* definida e com o a operação + definida a operação pode ser realizada;

```
3 t1.x = 20 t1.y = 30
4 t2.x = 10 t2.y = 5
5 t1 = t1 + t2
6 print (t1.x, t1.y) --> 30 35
```

Metamethods

- É possível modificar o comportamento da é o método *tostring*;
- Permite uma personalização de como deve ser impresso o objeto;

```
 2 meta.__tostring = function (a)
 3     return "x = " .. a.x .. ", y = " .. a.y
 4 end
 5
 6 t1.x = 20 t1.y = 30
 7 t2.x = 10 t2.y = 5
 8
 9 t1 = t1 + t2
10
11 print (t1) --> x = 30, y = 35
```

Metamethods

- Para tratar elementos não existentes em tabelas podemos usar os *metamethods index* e *newindex*;

- O *metamethod index* é chamado sempre que um campo ausente é acessado;

- *O metamethod newindex* é chamado sempre que atribuímos um valor a um campo ausente de uma tabela.

Metamethods


```lua
meta = {}
t = {}

meta.__index = function (self,k)
    print ("Acessando elemento não inicializado " .. k)
    return 0
end

setmetatable(t,meta)
print(t.x)              -->Acessando elemento não inicializado x
```

Metamethods


```
 2 t1 = {x=20, y=5}
 3 t2 = {x=30} -- não definimos um valor para o campo "y" de t2
 4
 5 setmetatable (t1, meta)
 6 setmetatable (t2, meta)
 7
 8 t1 = t1 + t2 --> Acessando elemento não inicializado y
 9 -- Quando tentamos acessar t2.y, o meta-metodo index retorna 0
10
11 print (t1) --> x = 50, y = 5
```

Metamethods

- O nome de um *metamethod* é precedido de dois traços __add, __sub, etc.

- A tabela a seguir apresenta os *metamethods* existentes em Lua

add	sub	mul	div	mod	pow
unm	idiv	band	bor	bxor	bnot
shl	shr	concat	len	eq	it
le	index	newindex	call	gc	mode
metatable	tostring	tonumber			

Orientação a Objetos

- Lua não é uma linguagem OO mas provê mecanismos que simulam;

- A ideia central da POO em Lua é o uso de *metatables* e *metamethods*.

- Utilizar um modelo OO onde o campo *index* da *metatable* será a própria *metatable*.

Orientação a Objetos

- Quando o campo *index* é uma tabela, Lua tenta acessar a chave ausente nessa tabela.

- Dessa forma, quando um método não for implementado por um objeto, iremos procurar pela implementação dele na sua *metatable*.

Orientação a Objetos

- Declarando uma classe:


```lua
--> Declarando classe Pessoa
Pessoa = {}
Pessoa.__index = Pessoa  --Associando metatable de Pessoa

-- Construtor de uma instancia de Pessoa
function Pessoa.create(codigo, nome)
    local p = {}
    setmetatable(p,Pessoa) --> Associando metatable de Pessoa a nova instancia da classe

    --Atributos da classe Pessoa
    p.codigo = codigo
    p.nome = nome
    return p
end
```

Orientação a Objetos

- Métodos:


```lua
function Pessoa:getNome()  --Retorna o nome
    return self.nome
end

function Pessoa.getCodigo(self)  --Retorna o codigo
    return self.codigo
end
```

Orientação a Objetos

- Herança simples


```lua
Cliente = Pessoa.create()  --Modo de fazer cliente herdar de Pessoa
Cliente.__index = Cliente  --Associando metatble de Cliente
--> Construtor de uma instancia de Cliente

function Cliente.create(codigo, nome, telefone)
    local c = Pessoa.create(codigo, nome) -- Cria nova instancia de Pessoa

    setmetatable(c,Cliente) --> Associando metatable de Cliente a instancia de Pessoa
    -- Atributos da classe Cliente
    c.telefone = telefone
    return c
end
```

Orientação a Objetos


```lua
function Cliente:__tostring()
    return "Codigo: " .. self.codigo .. " Nome: " .. self.nome .. " Telefone: ".. telefone
end

k = Cliente.create(1,"Jose","3333-3333")
print(k)

Codigo: 1 Nome: Jose Telefone: 3333-3333
```

Orientação a Objetos

- É possível fazer herança múltipla, porém é um pouco complicado;

- A ideia é que a função na chave __*index* da *metatabela* procure a chave em cada uma das classes pai, retornando a primeira ocorrência que encontrar;

- Coloca-se a lista de classes pai em outra chave da *metatable*.

Orientação a Objetos


```lua
Button = {
    super = { Shape, EventResponser }
}

Button.__index = function (t, k)
    local class
    for _, class in ipairs(Button.super) do
        if class[k] then return class[k] end
    end
end
```

Orientação a Objetos

- Para realizar polimorfismo e acrescentar novos métodos a esta classe modifica-se um pouco a função para que procure a chave primeiro na própria classe

```
1 Button.__index = function (t, k)
2     if Button[k] then return Button[k] end
3     local class
4     for _, class in ipairs(Button.super) do
5         if class[k] then return class[k] end
6     end
7 end
```

I/O

- Lua possui uma biblioteca com várias funções de I/O


```
boom = io.read () -- espera usuario digitar algo
print ( boom )
io.write (" bacon is overrated \n")
file = io.open (" arq.txt ","r")

--le a primeira linha de file
print ( file : read ())
file : close () --fecha o arquivo
file2 = io.open (" arq2.txt ","w")

-- escreve em file
file2 : write (" Long live the new flesh ")
file2 : close ()
```

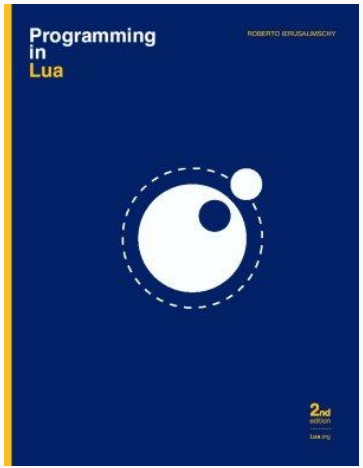
Modularização

- O sistema de módulos de Lua foi padronizado com o surgimento da versão 5.1, em 2006.

- Embora módulos já existissem e fossem usados há algum tempo pelos desenvolvedores, não existia ainda uma política bem definida para módulos.

Modularização

- Pela definição de módulo de *Programming in Lua(2ªed.)*
  - “Um módulo é uma biblioteca que pode ser carregada usando-se *require* e que define um único nome global que armazena uma tabela.“



- Todas as funções e variáveis do módulo estão contidas em uma tabela.

- No fundo módulos são tabelas;

Modularização

- É possível criar uma variável que guarda a referência para o módulo;

```
1 local math = require 'math'
```

- Podemos também armazenar as funções que um módulo exporta em variáveis locais:

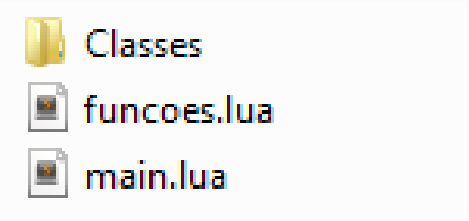
```
3 require 'math' --carrega modulo math
4 local sin = math.sin
5 local sqrt = math.sqrt
```

Modularização

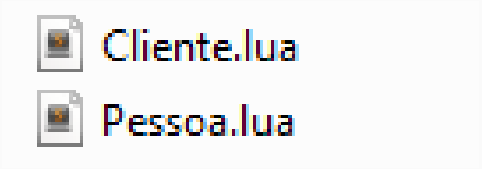
- Importando seu módulo de uma pasta do diretório atual:

```
1 require "Classes.Pessoa"
2 require "Classes.Cliente"
3 require "funcoes"
```

Diretório Raiz



Diretório Classes



Modularização

- Os seguintes módulos fazem parte da biblioteca padrão de Lua:
  - **coroutine:** possui as operações relacionadas com co-rotinas;
  - **string:** contem funções que manipulam cadeias de caracteres;
  - **table:** manipulação de tabelas;
  - **math:** modulo com as funções matemáticas;
  - **io:** biblioteca de entrada e sada (E/S);
  - **package:** biblioteca de módulos.
  - **os:** implementa facilidades do sistema operacional;
  - **debug:** biblioteca de depuração.

Modularização

- O interpretador já carrega os módulos da biblioteca padrão não sendo necessário fazer *require*;

- Uma maneira básica de instalar um módulo e simplesmente copiando o arquivo .lua, .so ou .dll para um diretório onde o interpretador de Lua poderá encontrá-lo.

- Outra forma é modificando o *LUA_PATH* colocando o caminho onde se encontra o módulo;

Bibliotecas Padrões

- Oferecem funções úteis que são implementadas diretamente através da API C;
- Oferecem serviços essenciais para a linguagem (e.g. *type* e *getmetatable*);
- Oferecem acesso a serviços "externos";
- Outras poderiam ser implementadas em Lua mesmo, mas são bastante úteis ou possuem exigências de desempenho críticas que merecem uma implementação em C (e.g. *table.sort*).
- Todas as bibliotecas são implementadas através da API C oficial e são oferecidas como módulos C separados.

Bibliotecas Padrões

- Atualmente, Lua possui as seguintes bibliotecas padrão:
  - biblioteca básica(*print, pairs*);
  - biblioteca de co-rotinas(*create, yield, resume*);
  - biblioteca de pacotes;
  - manipulação de cadeias(*strings*);
  - manipulação de tabelas(*table*);
  - funções matemáticas(*sin, log, etc.*);
  - operações *bit* a *bit*;
  - entrada e saída(teclado, abertura de arquivos);
  - facilidades do sistema operacional(*clock, date)*;
  - facilidades de depuração(consultar valor de variável).

Polimorfismo

- Lua apresenta os seguintes tipos de polimorfismo
  - Coerção
  - Sobrecarga
  - Inclusão

Polimorfismo

- Coerção
  - Lua provê conversão automática entre valores *string* e *number* em tempo de execução.
  - Qualquer operação aritmética aplicada a uma cadeia tenta converter essa cadeia para um número;
  - Inversamente, sempre que um número é usado onde uma cadeia é esperada, o número é convertido em uma cadeia, em um formato razoável.

Polimorfismo

- Coerção


```
1 print(2 + “3”) --> 5.0
2 print(“201” .. 5) --> 2015
3 print(“10”+”10”) --> 20.0
4 print(2 + “oi”) -->erro
5 print(“casa”+”rua”) --> erro
```

Polimorfismo

- Sobrecarga
  - Não possui sobrecarga de funções;
    - Porém, pode-se fazer uma “gambiarra” com a função *type,* analisando o tipo dos parâmetros
  - Sobrecarga de operadores é feita por *metamethods;*

Polimorfismo

- Sobrecarga


```lua
function overload (arg1 , arg2 )
    if type ( arg1 ) == 'string ' and type ( arg2 ) == 'string ' then
        return arg1 .. arg2
    elseif type ( arg1 ) == 'number ' and type ( arg2 ) == 'number ' then
        return arg1 + arg2
    end
end

a = overload ("10","10") --> 1010
b = overload (10 ,10) --> 20
```

Polimorfismo

- Inclusão
  - Permite a implementação de herança de classes;
  - Herança simples ou múltipla

Concorrência

- Não possui suporte à *threads;*
- Implementa co-rotinas, que são um poderoso mecanismo de programação em Lua;
- Às vezes confundida com função ou *thread*;
- É semelhante a uma thread, no sentido de que temos uma linha de execução compartilhando sua pilha de execução com outras co-rotinas.

Concorrência

- A grande diferença entre uma co-rotina e uma função é que a execução de uma co-rotina pode ser suspensa e retomada posteriormente (no ponto em que foi suspensa).

- A diferença entre co-rotinas e threads é que, conceitualmente, diferentes threads executam simulataneamente, enquanto que num sistema com co-rotinas, apenas uma co-rotina executa por vez.

Concorrência

- As funções que manipulam co-rotinas estão agrupadas na tabela *coroutine*.
- Uma co-rotina é criada passando uma função(em geral, anônima) para a função de criação, que retorna um valor do tipo thread;

```
2 co = coroutine.create( function ()
3                             print ("hi")
4                        end )
5
6 print (co) -> thread: 0x8071d98
```

Concorrência

- Pode estar em três estados:
  - Suspensa(*Suspended)*;
  - Executando(*Running*);
  - Morta(*Dead)*.
- Imediatamente após a sua criação, uma co-rotina está no estado “suspensa”.
- A função *coroutine.status* retorna o estado da co-rotina

```
3 print(coroutine.status(co))    --> suspended
```

Concorrência

- Para executar uma co-rotina que foi criada, invocamos a função *coroutine.resume*.

- Sua execução começa pela execução da função passada como parâmetro na sua criação.

```
2 coroutine.resume(co)   --> hi
3 coroutine.status(co)   --> dead
4 print(coroutine.resume(co)) --> false   cannot resume dead coroutine
```

Concorrência

- Pode-se suspender sua execução invocando a função *coroutine.yield.*

- Ao executar essa função, o controle volta para o código que tinha dado *coroutine.resume* na co-rotina, restaurando todo o ambiente local.

Concorrência

- Lua oferece um mecanismo simples e versátil para troca de dados (mensagens) entre co-rotinas.

- Os argumentos de uma chamada a coroutine.yield são passados como valores de retorno da chamada a coroutine.resume.

- Simetricamente, os argumentos de coroutine.resume são passados como valores de retorno da função coroutine.yield

Concorrência


```
2 co = coroutine.create(function (a,b,c)
3                           print("co", a,b,c)
4                       end)
5 coroutine.resume(co, 1, 2, 3)    --> co  1  2  3
```

Concorrência

- Uma chamada *resume* retorna, depois do sinal *true(*que significa que não houve erros) todos os argumentos passados pelo correspondente *yield;*

```
2 co = coroutine.create(function (a,b)
3                           coroutine.yield(a + b, a - b)
4                       end)
5 print(coroutine.resume(co, 20, 10))  --> true  30  10
```

Concorrência

- Simetricamente, *yield* retorna todos os argumentos extras passados pelo *resume* correspondente:


```
2 co = coroutine.create (function ()
3                            print("co", coroutine.yield())
4                        end)
5 coroutine.resume(co)
6 coroutine.resume(co, 4, 5)      --> co  4  5
```

Concorrência

- Finalmente, quando a co-rotina termina, todos os valores retornados pela função que foi executada são passados para o respectivo *resume;*


```
3 co = coroutine.create(function ()
4                           return 6, 7
5                       end)
6 print(coroutine.resume(co))    --> true  6  7
```

Concorrência

- As co-rotinas fornecidas por Lua são chamadas “Co-rotinas Assimétricas”

- Significa que existe uma função para suspender a execução de uma co-rotina e uma para retomar uma co-rotina suspensa;

- Diferentemente, das “Co-rotinas Simétricas” que possuem somente uma função para transferir o controle de uma co-rotina para a outra

Exceções

- Erros durante a execução do programa causam o fim da execução;
- Pode-se utilizar as funções *pcall* e *error* para simular um tratamento para os erros.
- Lança-se uma exceção com *error* e captura-se com *pcall*, a mensagem de erro identifica o tipo do erro.
- Não é obrigatório realizar o tratamento.

Exceções

- Necessário encapsular o código em uma função;
- Funciona de modo semelhante ao bloco *try/catch;*


```
2 function try () -->simulando bloco try
3             ...
4           if unexpected_condition then error() end
5             ...
6           print(a[i])    -- potential error: `a' may not be a table
7             ...
8 end
```

Exceções

- A função *pcall:*
  - Executa a função em modo protegido;
  - Captura algum erro durante a execução;
  - Se não ocorrer erro retorna *true* mais qualquer outro valor retornado;
  - Se ocorrer, retorna *false* mais a mensagem de erro;

```
2 if pcall(foo) then
3         ... -- no errors while running `foo'
4 else
5         ... -- `foo' raised an error: take appropriate actions
6 end
```

```
2 status, erro = pcall(foo) --> status recebe true ou false
3                           --> erro recebe a mensagem de erro caso ocorra
```

Exceções

- Pode-se utilizar *pcall* com funções anônimas
- O argumento de *error* pode ser de qualquer tipo


```
2 if pcall(function () ... end) then ...
3     else ...
4
5 local status, err = pcall(function () error({code=121}) end)
6
7 print(err.code)  -->  121
```

Constantes e Serialização

- Constantes
  - Lua não possui mecanismos para se definir constantes.

- Serialização
  - Lua não provê mecanismos para serialização na linguagem.
  - No entanto, existem várias funções criadas e fornecidas pela comunidade.
  - http://lua-users.org/wiki/TableSerialization

Gerenciamento de memória

- O gerenciamento de memória e realizado pelo coletor de lixo.
- O coletor libera a memória alocada por todos os objetos que deixaram de ser referenciados.
- Toda memória usada por Lua está sujeita ao gerenciamento automático;
- Utiliza o algoritmo marca-e-varre (*mark-and-sweep*) incremental.
- É possível implementar o *metamethod gc,* que marca o objeto para ser coletado

Gerenciamento de memória

- Usa dois números para controlar seus ciclos de coleta de lixo:

- **A *pausa do coletor de lixo***
  - Controla quanto tempo o coletor espera antes de começar um novo ciclo.
- ***Multiplicador de passo do coletor de lixo***
  - Controla a velocidade relativa do coletor em relação à alocação de memória.

- Você pode mudar esses números com a função *collectgarbage*.

Avaliação da Linguagem

Critérios Gerais	Lua	Java	C
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Parcial	Sim	Não
Aprendizado	Sim	Não	Não
Eficiência	Parcial	Parcial	Sim
Portabilidade	Sim	Sim	Não
Paradigma	Estruturado/ ”OO”	OO	Estruturado
Evolutibilidade	Sim	Sim	Não
Reusabilidade	Sim	Sim	Parcial
Interação	Sim	Parcial	Sim
Custo	?	?	?

Avaliação da Linguagem

Critérios Específicos	Lua	Java	C
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim
Gerenciamento de Memória	Sistema	Sistema	Programador
Persistência de Dados	Não	Serialização	Bibliotecas
Passagem de Parâmetros	Referência e Cópia	Referência e Cópia	Referência e Cópia

Avaliação da Linguagem

Critérios Específicos	Lua	Java	C
Encapsulamento e Proteção	Não	Sim	Parcial
Sistemas de Tipos	Sim	Sim	Não
Verificação de Tipos	Dinâmica	Estática / Dinâmica	Estática
Polimorfismo	Coerção/ Sobrecarga/ Inclusão	Todos	Coerção e Sobrecarga
Exceções	Não	Sim	Não
Concorrência	Não	Sim	Biblioteca de Funções

Referências

http://www.lua.org/pil/8.4.html

http://www.lua.org/manual/5.2/pt/

www.fabricadigital.com.br/media/**Curso_Lua**.pdf

www.**lua**.org/doc/wjogos04.**pdf**

http://pt.scribd.com/doc/66264884/Linguagem-Lua#scribd

http://montegasppa.blogspot.com.br/2007/02/orientao-objetos-em-lua.html

http://kodumaro.blogspot.com.br/2007/03/herana-mltipla-em-lua.html

http://lua-users.org/wiki/InheritanceTutorial