Lua

“Simple things simple, complex things possible”

André Martinelli, Breno Krohling, Israel Santos, Matheus Vieira, Pedro Henrique Flores

Introdução

Histórico

- Desenvolvida pelo grupo Tecgraf da PUC-Rio em 1993.
- Objetivo inicial de atender a um projeto da Petrobras.
- Influenciada pelas linguagens Lisp, Scheme
- Open Source, distribuída sob a licença MIT
- Versão atual 5.3.5
- Site: www.lua.org

Características

- Linguagem de script
- Multiparadigma
- Multiplataforma
- Tipagem dinâmica
- Simples e eficiente
  - Tabela é o único tipo de estrutura de dados
- Pequeno tamanho
  - aproximadamente 200 k
- Híbrida
- Coletor de lixo

Portabilidade

- Roda em muitas plataformas que já ouvimos falar
  - Unix, Windows, Symbiam, Palm, PSP, etc

- Escrita em ANSI C e ANSI C++

- Acoplada em C/C++, Java, Fortran, C#, Perl, Ada, etc
  - Biblioteca

Quem usa Lua?

- Adobe Photoshop Lightroom
- Apache HTTP Server
- Cisco Systems
- Wireshark
- VLC Media Player
- Desenvolvimento de Jogos

Getting Started

- Quatro maneiras de executar

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

```
$ lua <nome-do-arquivo.lua>
```

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

```
Lua 5.1.5  Copyright (C) 1994-2012 Lua.org, PUC-Rio
> print("Hello World")
Hello World
>
```

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

Sintaxe básica

```lua
--[[Comente sua função
aqui]]--
function olaMundo( )
    print('ola mundo')
end

local variavel = true

if(variavel) then
    olaMundo()
end
```

- Delimitadores de blocos por palavras chave
- Em Lua, variáveis são por padrão globais.

Sintaxe básica

```
1   a = 1
2   b = a*2
3
4   a = 1
5   b = a*2;
6
7   a = 1; b = a*2
8   a = 1 b = a*2
```

- Não é necessário usar “;” entre duas atribuições consecutivas, porém recomendável.

Declaração

```
1   print (b) -->nil
2   b = 10
3   print (b) --> 10
4
5   b = nil
6   print(b) --> nil
```

- Variáveis globais não precisam ser declaradas
- Simplesmente atribui-se um valor à variável
- Para deletar : “= nil”

Comentários

- Linha comentada “ -- “
- Bloco comentado “--[[<*code*>]]

```
--[[
    a = b
    print("bloco de")
    print("comentários")
]]
print ("Hello") --linha comentada
```

Palavras chave

- Lua possui as seguintes palavras chave

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

Amarrações

Identificadores

- Identificadores de variáveis em Lua podem conter números, letras e underscore, mas não podem começar com dígitos
- Variável não possui tamanho máximo de caracteres
- Case sensitive

```lua
local _a_ = 2
_A_ = 3
print(_a_, _A_)    -->2, 3

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

```lua
local _a2@_ = 2          --> erro
local 1inicio = 3        -->erro

a = 2; c =
3;          --> comando valido

```

Ambientes de amarração

- Escopo global por padrão, mesmo em ambientes locais e funções
- Palavra reservada *local* para escopo local

```lua
if(true) then
    variavelGlobal = "criada em escopo local"
end

print(variavelGlobal)  -->criada em escopo local

if (true) then
    local variavelLocal = "criada em escopo local"
end

print(variavelLocal)  -->nil
```

Ambientes de amarração

```
1  global =  "sou global"
2
3  if(true) then
4      local global = "assim escopo global não é visivel"
5      print (global)  --> assim escopo global não é visível
6  end
7  print (global)  --> sou global
```

- Neste caso a variável local é quem será chamada na função print dentro do escopo do comando if.
- Variáveis locais não são amarradas no início do bloco.

Ambientes de amarração

- Múltiplas atribuições

```
1   a, b = 10, "lua" --a = 10  b = "lua"
2   a, b, c = 0,1   -- c = nil
3   a, b = a+1, b+1, b+2 --b+2 is ignored
4   a,b, c = 0
5   print(a,b,c) -- 0 nil nil
```

Variáveis e constantes

Variáveis

- Baseadas no escopo
  - Global
  - Local
  - Tabela

Coletor de lixo

- Coletor de lixo nativo
- Mark and sweep

Valores e tipos de dados

Tipos de Dados

```
1  numero = 34
2  print(type(numero))  -->number
3
4  numero = "texto qualquer"
5  print(type(numero)) --> string
6
7  numero = print
8  numero(type(numero)) --> function
```

Perigoso, não façam isso em casa!

- Tipagem dinâmica

Tipos de Dados

- Possui os seguintes tipos
  - nil
  - boolean
  - number
  - string
  - function
  - userdata
  - thread
  - table

Tipos de Dados

```lua
print(type("Qual é o meu tipo?")) -->String
print(type(t)) --> nil ('t' não foi inicializado)
t = 10
print(type(5.8*t)) --> number
print(type(true)) --> boolean
print(type(print)) -->function
print(type(nil)) -->nil
```

- Variáveis apontam para nil antes de inicializadas.
- Número zero e string vazia são considerados como true em operações booleanas

Tipos de Dados

- nil
  - Ausência de valor
  - variável ainda não declarada
  - variável local declarada mas sem valor atribuído ainda
- boolean
  - 2 tipos: false e true
  - false: false e nil
  - Número zero e “vazio” são considerados como true em operações booleanas

```lua
b = false
if(b) then print("verdade") else print("falso") end --falso

b = true
if(b) then print("verdade") else print("falso") end --verdade

b = nil
if(b) then print("verdade") else print("falso") end --falso

b = "uma string legal"
if(b) then print("verdade") else print("falso") end --verdade

b = 0
if(b) then print("verdade") else print("falso") end -- verdade

b = {}
if(b) then print("verdade") else print("falso") end --verdade
```

Tipos de Dados

- number
  - Tipo Real (double-precision floating-point)
  - 32 bits
  - Não possui tipo Integer
- string
  - Imutáveis
  - Pode conter uma simples letra ou um livro inteiro
  - Tabela ASCII -> “\(*código*)” `print("\100")`
  - Caracteres especiais como em C: (\n , \t, \”, \b, etc)

```
> print("umalinha\npróxima\n\"aspas\" fim")
umalinha
próxima
"aspas" fim
```

Tipos de Dados

- string
  - conversão automática de string para number

```
1   print("10"+1) --11
2   print("10+1") -- 10+1
3   print("10"*"2") --20
4   print("hello"+1) --Erro!
```

  - number> string : concatenação (..) ou tostring()

```
1   print("Lua é "..10) --Lua é 10
2   b = tostring(10) --number to string
3   c = 10 .. ""
4   print(c) --10
5   print(type(c)) --string
```

Tabelas

- Única estrutura de dados disponível
- Utilizada para criar arrays, listas e dicionários
- Tamanho sob demanda

```lua
--inicialização
mytable = {}

--atribuição
mytable[1]= "Lua"

--removendo referencia
mytable = nil
```

Tabelas

- Construtores
  - Expressões que criam e inicializam tabelas
  - Permite a opção de palavras chaves
  - Lua sempre permite adicionar e remover campos e variáveis

```
1   tab = { } -- tabela vazia
2
3   days = {"Sunday", "Monday", "Tuesday"} --Lista
4   print(days[2])  --Monday
5
6   a = {x = 10, y = 20} --Conjuntos
7   print(a.x)  -- 10
8   a = { } ; a.x = 10; a.y = 20
9
.0  b = { [“lua”] = 3, [“C”] = 1, [“C++”] = 2}
.1  print(b[1])          -- nil
.2  print(b[“lua”])      -- 3
.3
.4  b[“lua”] = nil
.5  b[“java”] = 4
```

Tabelas

- Manipulando tabelas
  - concat: concatena os valores da tabela
  - insert: insere valor em posição especificada
  - maxn: retorna o maior index numérico
  - remove: remove valor da posição especificada
  - sort: ordena a tabela baseado em um parâmetro de comparação

- Concat

```lua
frutas = {"banana","laranja","abacaxi"}

-- concatena toda a tabela
print("string concatenada ",table.concat(frutas))


--concatena com string
print("string concatenada ",table.concat(frutas,", "))

--concatena por index
print("string concatenada ",table.concat(frutas,", ", 2,3))

--saida
--string concatenada    bananalaranjaabacaxi
--string concatenada    banana, laranja, abacaxi
--string concatenada    laranja, abacaxi

```

- insert e remove

```lua
frutas = {"banana","laranja","abacaxi"}

-- insere uma fruta no fim da tabela
table.insert(frutas,"manga")
print("Fruta na posicao 4 eh ",frutas[4])

--insere fruta na posicao 2
table.insert(frutas,2,"jambo")
print("Fruta na posicao 2 eh ",frutas[2])

print("O numero de elementos na tabela eh",table.maxn(frutas))

print("O ultimo elemento eh",frutas[5])


table.remove(frutas)
print("ultimo elemento foi removido?",frutas[5])

--saida
--Fruta na posicao 4 eh     manga
--Fruta na posicao 2 eh     jambo
--O numero de elementos na tabela eh    5
--O ultimo elemento eh  manga
--ultimo elemento foi removido? nil
```

- sort

```lua
frutas = {"banana","laranja","abacaxi","jambo"}

for k,v in ipairs(frutas) do
   print(k,v)
end

table.sort(frutas)
print("tabela ordenada")

for k,v in ipairs(frutas) do
   print(k,v)
end

--saida
--1 banana
--2 laranja
--3 abacaxi
--4 jambo
--tabela ordenada
--1 abacaxi
--2 banana
--3 jambo
--4 laranja
```

```lua
function ordena(a,b)
    return b < a
end

frutas = {"banana","laranja","abacaxi","jambo"}

for k,v in ipairs(frutas) do
   print(k,v)
end

table.sort(frutas, ordena)
print("tabela ordenada")

for k,v in ipairs(frutas) do
   print(k,v)
end
--saida
--1 banana
--2 laranja
--3 abacaxi
--4 jambo
--tabela ordenada
--1 laranja
--2 jambo
--3 banana
--4 abacaxi
```

arrays

- Tabelas indexadas por inteiros
  - Começa com o índice “1”

```lua
array = {"Lua", "Tutorial"}

for i = 0, 2 do
   print(array[i])      -->nil
end                     -->Lua
                        -->Tutorial
```

arrays

- index negativo

```
1  array = {}
2
3  for i= -2, 2 do
4     array[i] = i *2
5  end
6
7  for i = -2,2 do
8     print(array[i])
9  end
10
11
```

```
-->-4
-->-2
-->0
-->2
-->4
```

Matrizes e arrays multidimensionais

- Array de Arrays (ou Tabela de Tabelas)
- Cada array pode ter um tamanho variado

```lua
--Matriz MxN
mt = {} --cria a matriz
for i=1,M do
    mt[i] = {}-- cria uma nova linha
    for j=1,N do
        mt[i][j] = 0 --insere um valor
    end
end
```

Expressões e Comandos

Operações

- Operações aritméticas
  - ‘+’ (adição) , ‘-’ (subtração), ‘*’ (multiplicação), ‘/’ (divisão), ‘^’ (exponenciação), ‘%’ (módulo)
  - operador unário ‘-’ (negação)

- Operações relacionais
  - `<  >  <=  >=  ==  ~=`
  - somente para tipos iguais
  - resulta em ***true*** ou ***false***

```
1  if(2<15) then print("menor") else print('maior') end -- menor
2  if("2"<"15") then print("menor") else print('maior') end -- maior
3  if(2<"15") then print("menor") else print('maior') end-- Erro!
```

Operadores

- Operadores Lógicos
  - **and**, **or**, **not**
  - AND
    - Se o primeiro argumento é falso, retorna **false**.
    - Se o primeiro argumento é verdadeiro, retorna o segundo
  - OR
    - Se o primeiro argumento é falso, retorna o segundo
    - Se o primeiro argumento é verdadeiro, retorna **true**
  - Curto-Circuito
    - Ex: (v[5] and v[5]==1)

```
1   print(4 and 5) --5
2   print(nil and 13) --nil
3   print(4 or 5) -- 4
4   print(false or 5) -- 5
```

Operadores

- Operadores Lógicos
  - Default:
    - No exemplo ao lado , se x não tiver nenhum valor atribuído, terá o valor ‘3’ (default).

```
x = x or 3 --x = 3

x = 5
x = x or 3 -- x = 5
```

  - Operador Ternário:
    - Lua não tem operador ternário, mas pode-se construir um através dos operadores lógicos.
    - Ex: max = (x>y)?x:y

```
x = 3
y = 5
max = (x>y) and x or y
print(max) --5
```

Operadores

- Tamanho : ‘#’
  - Strings e Tables

```lua
a = "lp_lua"
print(#a) -- 6
t = {"um", "dois", "três"}
print((#t)) -- 3
```

- Prioridade (cima para baixo)

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

While

- Número indeterminado de repetições

```
1 a = 10
2
3 while( a < 20 )
4 do
5    print("valor de a:", a)
6    a = a+1
7 end
8
```

for

- Número determinado de repetições

```
1
2 for i = 10,1,-1
3 do
4     print(i)
5 end
```

repeat..until

- Número indeterminado de repetições, porém com checagem de condição no final

```
1
2 a = 10
3
4
5 repeat
6    print("valor de a:", a)
7    a = a + 1
8 until( a > 15 )
```

Desvios

- goto (suportado somente pela última versão)
- break
- return

I/O

- Operações implícitas e explícitas
  - implícitas define um arquivo como padrão para qualquer operação io
  - explícitas o arquivo chama funções de uma metatable, podendo ter mais de um arquivo aberto e manipulado
- Possível abrir arquivos em modo de leitura, escrita e “adicionar texto”
- Algumas funções disponíveis
  - **io.open()**
  - **io.read()**
  - **io.write()**
  - **io.flush()**
  - **io.lines()**
  - **file:seek()**

I/O

- Implícitas

```lua
-- Abre arquivo em modo de leitura
file = io.open("teste.lua", "r")

-- define arquivo teste.lua como input padrao
io.input(file)

-- printa primeira linha
print(io.read())

-- fecha arquivo
io.close(file)

-- Abre arquivo em mode de "adicao"
file = io.open("teste.lua", "a")

-- define arquivo teste.lua como output padrao
io.output(file)

-- Adiciona uma string no final do arquivo
io.write("fim do teste")

-- fecha o arquivo
io.close(file)
```

I/O

- Explícita

```lua
-- Abre arquivo em modo de leitura
file = io.open("teste.lua", "r")

-- printa primeira linha
print(file:read())

-- fecha arquivo
file:close()

-- Abre arquivo em mode de "adicao"
file = io.open("teste.lua", "a")

-- Adiciona uma string no final do arquivo
file:write("fim do teste")

-- fecha o arquivo
file:close()
```

Modularização

Funções

- Funções são declaradas utilizando a palavra chave *function*

```
1 function minhaFuncao(var)
2     print(var)
3 end
4
5 minhaFuncao("utilizando uma funcao") -->utilizando uma funcao
```

Funções

- Lua permite múltiplo retorno nas suas funções

```lua
function retornoMultiplo()
    return "um", 2, true
end

a,b,c = retornoMultiplo()

print(a,b,c)  -->um 2 true

a,_,c = retornoMultiplo()
print(a,c)  -->um true

a,b,c,d = retornoMultiplo()
print(a,b,c,d)  --> um 2 true nil
```

Funções

- Funções aceitam número variável de parâmetros

```lua
function media(...)
    resultado = 0
    local arg = {...}
    for i,v in ipairs(arg) do
        resultado = resultado + v
    end
    return resultado/#arg
end

print("A media eh",media(10,5,3,4,5,6))
```

Funções

- Parâmetros default

```lua
function soma3numeros(a,b,c)
    a = a or 1
    b = b or 1
    c = c or 1

    return a+b+c
end

print(soma3numeros(2,2,2)) -->6
print(soma3numeros(2,_,2)) -->5
print(soma3numeros())      -->3
```

Funções

- Parâmetros por nome

```lua
function myfunction(t)
    setmetatable(t,{__index={a=1, b=7, c=5}})
    local a, b, c =
      t[1] or t.a,
      t[2] or t.b,
      t[3] or t.c

    print(a,b,c)
end

myfunction{} -->1,7,5
myfunction{2,3,4} -->2,3,4
myfunction{2,c=8, b=3} -->2,3,8
```

Funções

- Passagem de parâmetros é posicional e por cópia para variáveis que não são tabelas

```lua
function parametros(a,b)
   a = 1
   print(a,b)        -->1 3
end

local a = 2
local b = 3

parametros(a,b)
print(a)      --> 2
```

Funções

- Quando passamos uma tabela, é feita passagem por referência. Assim podemos alterar os valores da tabela na função.

```lua
function parametros(a)
    a[1] = 1
    print(a[1],a[2])      -->1 3
end

local a={2,3}

parametros(a)
print(a[1])      --> 1
```

Módulos

- Módulos em Lua são como pacotes.
- Variável global contendo uma tabela
- Funções e variáveis do módulo são inseridos na tabela
- Palavra chave *require*

Módulos

exemplos.lua | mymath.lua

```lua
meuModulo = require("mymath")
meuModulo.add(10,20) -->30
meuModulo.sub(30,20) -->10
meuModulo.mul(10,20) -->200
meuModulo.div(30,20) -->1.5
```

exemplos.lua | mymath.lua

```lua
local mymath =  {}

function mymath.add(a,b)
   print(a+b)
end

function mymath.sub(a,b)
   print(a-b)
end

function mymath.mul(a,b)
   print(a*b)
end

function mymath.div(a,b)
   print(a/b)
end

return mymath
```

Iterators

Iterator

- Um iterator é qualquer construção que permite percorrer os elementos de uma array por exemplo.
- Função ‘ipairs’
  - key = chave (ou índice)
  - value = elemento

```lua
array = {"Lua", "iter"}
for key, value in ipairs(array) do
    print(key, value)
end
-- 1     Lua
-- 2      iter
```

Iterator

- É possível criar o próprio iterator através de funções
  - ‘return’ várias vezes
  - Cada vez que a função é chamada ela retorna o próximo elemento

```
1   function square(maximo, numero)
2       if(numero<maximo) then
3           numero = numero+1
4           return numero, numero*numero
5       end
6   end
7
8   for i,n in square,3,0 do
9       print(i,n)
10  end
11  -- 1    1
12  -- 2    4
13  -- 3    9
```

Iterator

- Função next(array, key)
  - Retorna o próximo elemento a partir da chave ‘key’
  - ‘key’ também pode ser um índice
  - Se este campo estiver nulo, retorna o primeiro elemento

```lua
array{"Lua", "LP", "tutorial"}
print(next(array, 2)) --"tutorial"
print(next(array)) -- "Lua"

for k,v in next, array do
    print(k,v)
end
-- 1  Lua
-- 2  LP
-- 3  tutorial
```

Iterator

- Exemplo:
  - criando uma lista encadeada
  - invertendo chaves e valores na tabela

```lua
list = nil
valores= {1, 2, 3,9 ,0,4}
for key, elem in ipairs(valores) do
    list = {val = elem, next = list}

end
```

Iterator

- Exemplo:
  - criando uma lista encadeada
  - invertendo chaves e valores na tabela

```lua
list = nil
valores= {1, 2, 3,9 ,0,4}
for key, elem in ipairs(valores) do
    list = {val = elem, next = list}

end
```

```lua
alunos = {["Joaquin"] = "Lua", ["Ana"]= "Python", ["Jonicley"] = "Java"}
linguagem ={}
print(alunos["Joaquin"]) -- Lua
for key,elem in pairs(alunos) do
    print (elem)
    linguagem[elem] = key
end
print(linguagem["Java"]) -- Jonicley
```

Compilação e Execução

Compilação

- De forma similar a Java, Lua ao ter seu código fonte compilado gera um *bytecode* que será interpretado por uma máquina virtual

- Embora Lua seja uma linguagem interpretada, é possível compilar funções, arquivos e códigos no próprio tempo de execução (*runtime)*.

- Principais funções:
  - *dofile, loadstring, loadfile*

Compilação

- Loadstring
  - Armazena uma linha de código
  - Ao chamar esta função, o código é executado
  - Deve ser utilizada com cuidado
    - Executa de forma lenta e pesada
  - Loadstring compila em um ambiente global
    - f = loadstring(“**local** a = 10; a = a+2”)

```
1   f = loadstring("i = i+1")
2   i = 0
3   f(); print(i) --1
4   f()
5   f(); print(i) --3
6
7   --Forma direta
8   s = "i = i+1"
9   loadstring(s)()
10  print(i) -- 4
```

Compilação

- Loadstring
  - Armazena uma linha de código
  - Ao chamar esta função, o código é executado
  - Deve ser utilizada com cuidado
    - Executa de forma lenta e pesada
  - Loadstring compila em um ambiente global
    - f = loadstring(“**local** a = 10; a = a+2”)

```
1  i = 32
2  local i= 0
3  f = loadstring("i = i+1; print(i)")
4  g = function () i = i+1; print(i) end
5  f() --33
6  g() --1
```

```
1  f = loadstring("i = i+1")
2  i = 0
3  f(); print(i) --1
4  f()
5  f(); print(i) --3
6
7  --Forma direta
8  s = "i = i+1"
9  loadstring(s)()
10 print(i) -- 4
```

Compilação

- Loadfile
  - Compila arquivos .lua
  - Compilando arquivos por esta função evita-se efeitos colaterais

Hello. lua

```lua
function imprime(x)
    print(x)
end
```

```
>
>
> f = loadfile("hello.lua")
> imprime("teste")
stdin:1: attempt to call global 'imprime' (a nil value)
stack traceback:
        stdin:1: in main chunk
        [C]: ?
> f()
> imprime("teste")
teste
>
```

Erros

Erros

- Como Lua é frequentemente empregada em aplicações, não se pode simplesmente abortar o programa na maioria das vezes.
- Condições inesperadas
  - Adicionando número e string
  - Utilizando variáveis inexistentes
  - Tabelas com índices inexistentes

```
> a= 3+"string"
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
        stdin:1: in main chunk
        [C]: ?
>
>
>
> array = {"Lua"}
> print(arrai[1])
stdin:1: attempt to index global 'arrai' (a nil value)
stack traceback:
        stdin:1: in main chunk
        [C]: ?
```

Erros

- Explicitando um erro
  - *error function*
    - argumento: “mensagem”

```
1  print "enter a number:"
2  n = io.read("*number")
3  if not n then error("eu quero numero!") end
```

```
enter a number:
Arnaldo
lua5.1: hello.lua:3: eu quero numero!
```

  - assert function
    - Confere se o primeiro argumento é verdadeiro e retorna-o
    - Caso seja falso, chama a função erro e a mensagem como argumento

```
1  print "enter a number:"
2  n = assert(io.read("*number"), "eu quero numero!")
```

Erros

- *Quando uma função encontra um erro*
  - *retorna nil*
  - *chama a função error(msg)*

- *Tratamento:*
  - *if not var then <<code>> end*

```
1  local file, msg
2  repeat
3      print "enter a file name:"
4      local name = io.read()
5      file, msg = io.open(name, "r")
6      --file = nil
7      if not file then print(msg) end
8  until file
```

```
enter a file name:
primeiro.txt
primeiro.txt: No such file or directory
```

Erros

- Tratamento de erro
  - assert function

```lua
local function add(a,b)
    assert(type(a)=='number', 'a is not a number')
    assert(type(b)=='number', 'b is not a number')
    return(a+b)
end

print(add(2,3)) --5
add(10) --b is not a number
```

Erros

- “throw an exception “ = error function
- “catch an exception” = pcall function

```lua
a = 3
b = "abc"
if pcall("c = a+b") then
    print("soma ok")
else
    print("soma errada")
end
-- soma errada
```

Erros

- Erros lançados em funções
  - não precisa declarar
- “throw an exception “
  - error function
- “call an exception”
  - pcall function

```lua
function foo
    --code
    if(not a) then --condição inesperada
        error() --lança o erro
    else
        print(a[1])
    end
end

if pcall(foo) then
    --no errors
else --captura um erro
    --<tratamento de erro>
end
```

Coroutines

Coroutines

- Linha de execução de uma função, com suas próprias variáveis locais e instruções.
- Entretanto, compartilham as variáveis globais.
- Apenas uma coroutine é executada por vez
- Table *coroutine*
- Tipo de dados: thread

Coroutines

- Create
  - Cria a coroutine
  - Argumento: maioria das vezes uma função anônima
- Resume
  - Reinicia a rotina caso esteja suspensa
  - Não é possível reiniciar depois de morta
- Status
  - Verifica o status
  - dead
  - running
  - normal
  - suspended

```lua
co = coroutine.create(function () print("hi") end)
print(co) -- thread: 0x2171170
--co() > error
print(coroutine.status(co)) --suspended
coroutine.resume(co) --hi
print(coroutine.status(co)) -- dead
coroutine.resume(co) -- não executa
```

Coroutines

- yield function
  - Permite suspender uma coroutine em execução
  - Posteriormente pode voltar do ponto onde parou (coroutine.resume)

```
1   co = coroutine.create(function ()
2       for i=1,4 do
3           print("co", i)
4           coroutine.yield()
5       end
6   end)
7
8   coroutine.resume(co) --co   1
9   coroutine.resume(co) --co   2
0   print(coroutine.status(co)) --suspended
1   coroutine.resume(co) --co   3
2   coroutine.resume(co) --co   4
3   print(coroutine.status(co)) -- suspended
4   coroutine.resume(co) --
5   print(coroutine.status(co)) --dead
```

Coroutines

- Passagem de parâmetros
  - Entrada (function)

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

coroutine.resume(co, 1,2,"lua") --co  1  2  lua
```

  - Saída (Yield)
    - false ou true
    - print function

```lua
co = coroutine.create(function(a,b)
    coroutine.yield(a+b, a-b)
end)
t,v,x = coroutine.resume(co, 3, 1)
print(t) -- true
print(v) -- 4
print(x) -- 2
```

Polimorfismo

Coerção

- Ad-hoc
  - Como Lua é dinamicamente tipada, não é feita a coerção em atribuições.
  - Provê conversão automática entre valores string e number em tempo de execução
  - Operações aritméticas aplicadas a uma cadeia tentam converter essa cadeia para um número.

```lua
numero = 10
string = "12"

soma = numero + string
concatenacao = numero..string

print(type(soma), soma)     --> number 22
print(type(concatenacao),   -->string 1012
   concatenacao)
```

Coerção

- Sobrecarga
  - possível utilizando mecanismo de metatabelas
  - tabelas possuem metatabelas, as quais podemos sobrescrever métodos

```lua
f1 = {a = 1, b = 2} --cria duas tabelas
f2 = {a = 2, b = 3}

--s = f1 + f2 //gera erro

metatabela = {}    --cria uma tabela

function metatabela.__add(f1,f2)    -->sobresvrece a funcao
    soma = {}                       --> add da metatabela
    soma.a = f1.a + f2.a
    soma.b = f1.b + f2.b
    return soma
end

setmetatable(f1,metatabela)     -->seta a metatabela como metatable
setmetatable(f2,metatabela)     --> de f1 e f2

s = f1 + f2                 -->chama __add na metatable de f1
print(s.a, s.b)  -->3,5
```

- Imprimir um tipo de dado
  - metamethod : __tostring
  - cria-se uma função para imprimir
    - retorna uma string

```lua
f = {a = 3, b = 5} --conjunto
mt = {} -- metatable
setmetatable(f, mt)
function f.imprime(x)
    return "Elemento a = ".. x.a .. "\nElemento b = ".. x.b
end
mt.__tostring = f.imprime
print(f) --Elemento a = 3
         --Elemento b = 5
```

- Algumas funções que podemos sobrescrever

```
-- __add(a, b)                     for a + b
-- __sub(a, b)                     for a - b
-- __mul(a, b)                     for a * b
-- __div(a, b)                     for a / b
-- __mod(a, b)                     for a % b
-- __pow(a, b)                     for a ^ b
-- __unm(a)                        for -a
-- __concat(a, b)                  for a .. b
-- __len(a)                        for #a
-- __eq(a, b)                      for a == b
-- __lt(a, b)                      for a < b
-- __le(a, b)                      for a <= b
-- __index(a, b)  <fn or a table>  for a.b
-- __newindex(a, b, c)             for a.b = c
-- __call(a, ...)                  for a(...)
```

Universal

- Paramétrico
  - Em Lua não precisamos declarar tipos de parâmetros e retorno, logo o polimorfismo paramétrico é feito diretamente pela linguagem

```lua
t = {}
table.insert(t,"10")
table.insert( t, 10 )
table.insert( t, function() return 10 end )

for k, v in pairs( t ) do
   print( type(v), v )  --> string 10
end                     --> number 10
                        -->function function: 0x12b9f90

```

Universal

- Inclusão
  - Lua, não é uma linguagem Orientada a Objetos, porém conseguimos simular isso através de tabelas.
  - Desse modo também conseguimos utilizar polimorfismo de inclusão, através de herança.

- Simulando orientação a objetos

~/Documentos/shape.lua - Sublime Text (UNREGISTERED)

main.lua | retangulo.lua | shape.lua

```lua
-- Meta class
local Shape = {area = 0}

-- Base class method new

function Shape:new (o,lado)
   o = o or {}
   setmetatable(o, self)
   self.__index = self
   lado = lado or 0
   self.area = lado*lado;
   return o
end

-- Base class method printArea

function Shape:printArea ()
   print("area =  ",self.area)
end

return Shape
```

~/Documentos/retangulo.lua - Sublime Text (UNREGISTERED)

main.lua | retangulo.lua | shape.lua | casa.lua

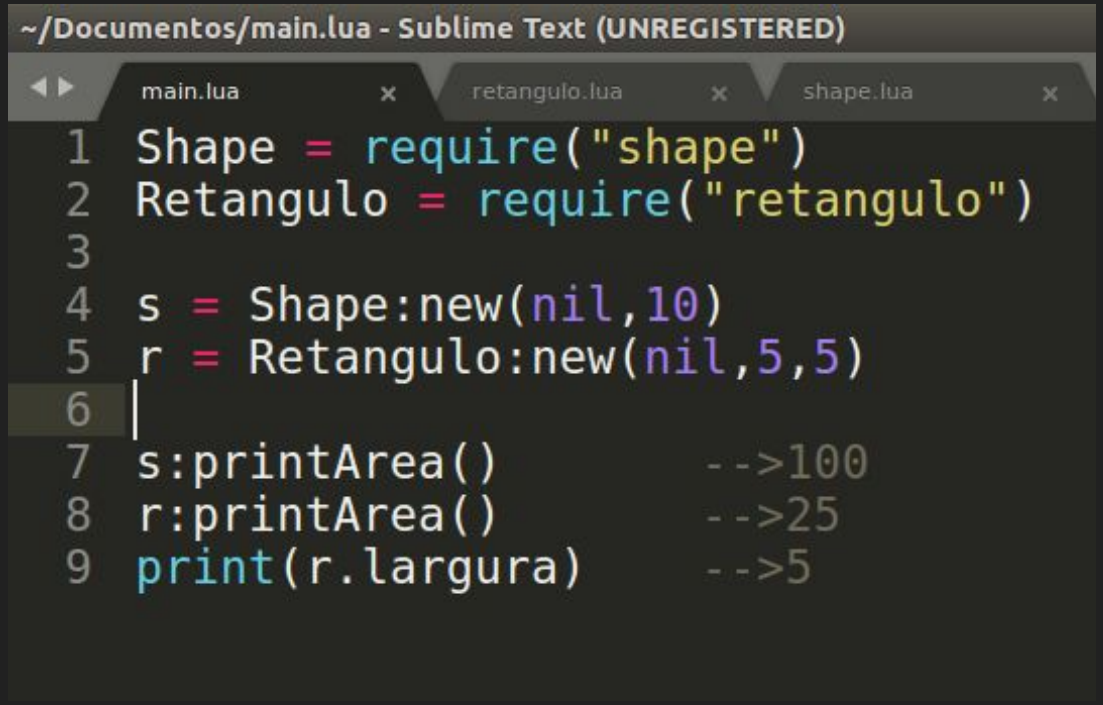
```lua
Retangulo = Shape:new()

-- metodo new

function Retangulo:new (o,altura,largura)
   o = o or Shape:new(o)
   setmetatable(o, self)
   self.__index = self
   self.area = altura * largura
   self.altura = altura
   self.largura = largura
   return o
end


return Retangulo
```

- Instanciando objetos



Outras Estrutura de Dados em Lua

Filas

- Funções *insert* e *remove* podem ser custosas para estruturas grandes
- 2 índices: primeiro e último elemento

```lua
--Retira o ultimo indice
function List.poplast (list)
    local last = list.last
    if list.first > last then
        error("fila vazia")
    end
    local value = list[last]
    list[last] = nil
    list.last = last - 1
    return value
end
```

```lua
List = {}
--Cria a Fila
function List.new ()
    return {first = 0, last = -1}
end
--Adiciona ao primeiro indice
function List.pushfirst (list, value)
    local first = list.first - 1
    list.first = first
    list[first] = value
end
--Adiciona ao ultimo indice
function List.pushlast (list, value)
    local last = list.last + 1
    list.last = last
    list[last] = value
end
--Retira o primeiro indice
function List.popfirst (list)
    local first = list.first
    if first > list.last then
        error("fila vazia")
    end
    local value = list[first]
    list[first] = nil
    list.first = first + 1
    return value
end
```

Conjuntos

- Exemplo: Conjunto de linguagens já apresentadas em LP

linguagens ={ [“python”] = true, [“scala”] = true, [“lua”] = true, [“javascript”] = true }

if(linguagens[“lua”]) then print(“linguagem já apresentada”) end

```
function Set.new(list)
    local conj  ={}
    for _, lp in ipairs(list) do
        conj[lp] = true
    end
    return conj
end

linguagens = Set.new{"Python", "Scala", "Lua", "Java"}
```

Union

- Lua não tem este tipo, porém pode-se implementar um método com os dois conjuntos.
- Exemplo: variável Set do exemplo anterior

```lua
function Set.union (a, b)
    local res = Set.new{}
    --adiciona os elementos do conjunto a
    for k in pairs(a) do res[k] = true end
    --adiciona os elementos do conjunto b
    for k in pairs(b) do res[k] = true end
    return res
end
```

- Utilizando metamethods também é possível

```lua
mt = {} --metatable
mt.__add = Set.union
setmetatable(a,mt)
setmetatable(b,mt)
c = a+b -- union
```

Lua como API

- Lua é organizada como uma biblioteca em C

- Exporta pouco menos de 100 funções
  - Executa trechos de códigos em Lua (scripts)
  - Chama e registra funções
  - Manipulação de tabelas

- O programa Lua desta biblioteca possui menos de 400 linhas de código

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Lua é uma biblioteca em C, e pode ser compilada em qualquer plataforma que tenha um compilador de C ou C++.		
Aprendizado	Não			
Eficiência	Sim			
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Possui tipagem dinâmica pode fazer com que o programador cometa muitos erros simples e difíceis de se identificar.		
Eficiência	Sim			
Portabilidade	Não			
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	“Simple things simple, complex things possible”.		
Portabilidade	Não			
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Possui alguns aspectos que diminuem a eficiência, por exemplo o coletor de lixo.		
Método de projeto	Estruturado	Estru		
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estru	É distribuída via um pequeno pacote e compila sem modificações em todas as plataformas que têm um compilador C padrão.	
Evolutibilidade	Não	P		
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	P	Ela permite programação procedural, programação orientada a objetos, programação funcional, programação orientada a dados e descrição de dados.	
Reusabilidade	Parcial			
Integração	Sim			
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Lua ainda recebe atualizações de uma equipe de desenvolvedores da PUC-Rio		
Integração	Sim			
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim			
Custo	Depende da ferramenta	Dep fer		

Lua é uma linguagem de extensão, outro programa cliente utiliza suas funções. Logo qualquer sistema compatível com Lua pode usar suas funções

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	P		
Reusabilidade	Parcial			
Integração	Sim	Sim	Parcial	Sim
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Lua trabalha acoplada a uma aplicação hospedeira que pode utilizar recursos fornecidos pela linguagem.

Critérios Gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Não
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial			
Integração	Sim			
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Como Lua é muito utilizada em conjunto com outras linguagens, irá depender do custo de outras aplicações

Critérios Gerais	C	C++	Java	Lua
Expressões e comandos	Sim	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim	Sim
Gerenciamento de memória	Programador	Programador	Sistema	Sistema
Persistência de dados	Sim	Sim	Sim	Não possui serialização fornecida pela linguagem
Passagem de parâmetro	Lista variável e por valor	Lista variável, por cópia, e referência	Lista variável, por cópia e cópia de referência	Lista variável, por cópia e referência
Exceções	Não	Sim	Sim	Não

Referências

- https://pt.wikipedia.org/wiki/Lua_(linguagem_de_programa%C3%A7%C3%A3o)
- https://www.lua.org/
- https://www.tutorialspoint.com/lua/
- https://stackoverflow.com/