Gabriel Correa de Macena
Marcus Vinicius Palassi Sales

Departamento de Informática
Graduação em Ciência da Computação

11 de junho de 2015

Sumário

Introdução

- A história de Lua
- Características
- Lua na atualidade

A história de Lua

A primeira linguagem de programação brasileira

- Criada em 1993 por Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes, membros do Tecgraf, o grupo Tecnológico de Computação Gráfica da Pontíficia Universidade Católica do Rio de Janeiro.
- Precedida pelas linguagens DEL (Data-Entry Language) e SOL (Simple Object Language), usadas para aplicações de engenharia para a Petrobras.
- Licenciada sob a licença MIT, a partir da versão 5.0.
- Influenciada principalmente por C++, Modula, Scheme, CLU e SNOBOL.

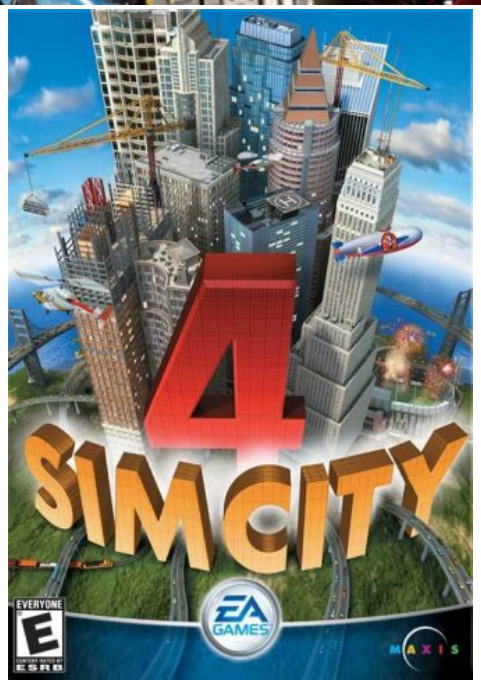
Características
A linguagem Lua

- É multi-paradigma. Suporta programação imperativa (tanto estruturada quanto orientação a objetos baseado em protótipos), além de possuir conceitos de programação funcional.
- É usada primariamente como linguagem de script.
- É escrita em ANSI C e portanto, é multiplataforma.
- Usa tipagem dinâmica e é fortemente tipada.

Lua na atualidade
Principais aplicações

- Lua é uma das principais linguagens de script usadas na indústria de jogos eletrônicos, por ser facilmente integrável, por sua rapidez e por sua curva de aprendizado plana.
- Em janeiro de 2012, Lua foi anunciada como vencedora do *Front Line Awards 2011* da revista *Game Developer* na categoria de ferramentas de programação.

Lua na atualidade
Principais aplicações



Lua na atualidade
Principais aplicações

Declarações de variáveis:

```
var1 = 999
var2 = "the dark side of the lua"
var3 = true
```

Lua permite múltipla atribuição:

```
a,b,c = 9.78,"goku",6.84
```

- Operadores aritméticos: $+$, $-$ (subtração e negação), $*$, $/$, $\%$
- Operadores relacionais: $<$, $>$, $<=$, $>=$, $==$, $\sim=$
- Operadores lógicos: **`and`**, **`or`** e **`not`**

Todos os operadores aritméticos operam em números reais.

```
p = 5 + 4.78
l = 111 - 11
q = 322 * 2
u = 16 / 4
r = 8^2
f = 42 % 8
t = -5.1
```

- Todos os operadores relacionais retornam **true** ou **false**.
- Os operadores $<$, $>$, $<=$ e $>=$ só podem ser usados com dois números ou com duas strings.
- Os operadores $==$ e $\sim=$ podem ser usados com valores de qualquer tipo. Mas comparações entre valores de tipos diferentes retornam sempre **false** e **true**, respectivamente.

```
a = 2 < 15 --> true
b = "2" < "15" --> false
c = 8.1 >= 7.5 --> true
d = 9.4 <= 10 --> true
e = 10 == "10" --> false
f = 145 ~= "145" --> true
```

Conceitos básicos
Operadores lógicos

- O operador **and** retorna seu primeiro argumento caso este seja **falso**, e retorna o segundo caso contrário.
- O operador **or** retorna seu primeiro argumento caso este seja **verdadeiro**, e retorna o segundo caso contrário.
- Ambos operadores usam avaliação de curto-circuito.
- O operador **not** retorna somente **true** ou **false**.

```
print(268 and 255) --> 255
print(false and "batman") --> false
print(471 or 399) --> 471
print(false or 6.38) --> 6.38
print(not false) --> true
print(not 0) --> false
```

- O operador **and** retorna seu primeiro argumento caso este seja **falso**, e retorna o segundo caso contrário.
- O operador **or** retorna seu primeiro argumento caso este seja **verdadeiro**, e retorna o segundo caso contrário.
- Ambos operadores usam avaliação de curto-circuito.
- O operador **not** retorna somente **true** ou **false**.
- Lua 5.3 adicionou operadores lógicos *bit* a *bit*: & (AND), | (OR), $\wedge$ (XOR) e $\sim$ (NOT).

```
print(268 and 255) --> 255
print(false and "batman") --> false
print(471 or 399) --> 471
print(false or 6.38) --> 6.38
print(not false) --> true
print(not 0) --> false
```

- **if-then-else**

```
if a < b then
  return a
elseif a > b
  return b
else
  return "sao iguais"
end
```

- **while**

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

- **repeat-until**

```
repeat
  line = io.read()
  print(line)
until line == ""
```

- **for**

```
--for numerico
for var = inicio,fim,incremento do
  --corpo
end

--for generico
--impressao de valores de um array
for i,v in pairs(a) do
  print(v)
end
```

- **break**

```
local i = 1
while a[i] do
  if a[i] == v then break end
  i = i + 1
end
```

- **return**

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

Cuidado!

Break* e **return** só podem ser usados como última declaração de um bloco.

Cuidado!

Break* e **return** só podem ser usados como última declaração de um bloco.

* Lua 5.2 permite que **break** também apareça no meio do bloco.

Lua 5.2 trouxe o desvio irrestrito à linguagem, com a adição da palavra reservada **goto**.

```
-- Lua 5.2.0-beta-rc2
::redo::
for x=1,10 do
  for y=1,10 do
    if not f(x,y) then
      goto continue end
    if not g(x,y) then
      goto skip end
    if not h(x,y) then
      goto redo end
    ::continue::
  end
end
::skip::
```

Conceitos básicos
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()
```

- Lua possui implementação híbrida. Um script em Lua é compilado em um *bytecode* que é executado na máquina virtual Lua.
- O processo de compilação é tipicamente invisível ao usuário, acontecendo durante o tempo de execução. Porém, também é possível somente compilar o script.

```
lua script.lua #interpretacao de script
luac script.lua #gera um bytecode luac.out
```

Identificadores

Identificadores em Lua

Identificadores em Lua são quaisquer combinações de letras, números e subtraços(_) que não começam com um número.

Exemplos

```
umaVariavel = 10 --valido
mais1Variavel = "uma string" --valido
_OUTRAVARIAVEL = "hala madrid" --valido
4Variavel = 322 --invalido
```

Identificadores

Sugestão

Roberto Ierusalimschy recomenda que variáveis que comecem com subtraço seguido de letras em caixa alta não sejam usadas, pois são reservadas para usos especiais pela linguagem.

Identificadores
Palavras reservadas

As seguintes palavras são reservadas em Lua:

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

Lua é case-sensitive

No entanto, por Lua ser case-sensitive, podemos ter por exemplo, **And**, **aNd**, **anD**, **AND** e variações como identificadores.

Identificadores
Comentários simples

- Comentários em Lua começam em qualquer parte do código com – (dois hífens).
- Lua oferece também comentários em bloco, que começam com –**[[** e vão até o primeiro **]]**.

Dica

Sempre coloque seus comentários em bloco entre –**[[** e –**]]**.

Exemplos

```
--print(42)
s = "nao estou comentado!"
--[[
a = 1
b = 2
c = 3
]]
t = "eu tambem nao!"
--[[
print(10)
--]]
```

Para criar uma variável global, a única coisa que devemos fazer é atribuir um valor à ela.

Exemplo

```
euSouUmaVariavelGlobal = 1437
```

O acesso a uma variável global não inicializada não gera erro, mas retorna o valor **nil**.

Exemplo

```
print(b) --> nil, b nao inicializado
b = 10
print(b) --> 10, b inicializado
```

Geralmente, não é necessário apagar variáveis globais. Caso uma variável tenha um período curto de vida, a recomendação é declará-la com a palavra-chave **local**.

Exemplo

```
local naoSouGlobal = "sou uma string"
```

Mas caso seja necessário apagar alguma variável global, devemos atribuir **nil** à variável.

Exemplo

```
b = nil
print(b) --> nil
```

Após essa atribuição, Lua se comporta como se a variável nunca tivesse sido usada. Em outras palavras,

Definição

Uma variável em Lua é *existente* se e somente se tem um valor diferente de **nil**.

- Lua usa somente escopo estático.
- Foi definido como padrão que toda variável não definida como **local**, é global.

Exemplo

```
function f(x)
  local var_local = 10 -- variavel local
  var_global = 11 -- variavel global
  print(var_local) -- 10
  print(var_global) -- 11
end

print(var_local) -- nil
print(var_global) -- 11
```

Escopo
Escopo em Lua

- Variáveis locais só existem no bloco onde foram declaradas.
- O interpretador sempre procura a última declaração de uma variável antes de executar uma operação com tal variável.

Exemplo

```
a = 5
print(a) --> 5
do
-- cria uma nova variavel a,
-- local ao bloco do
  local a = 6
  print(a) --> 6
end
print(a) --> 5
```

Valores e Tipos de Dados em Lua

- Por ser dinamicamente tipada, variáveis em Lua não possuem tipos, apenas valores têm tipos.
- Todos os valores em Lua são ditos de *primeira classe*, ou seja, podem ser atribuídos à variáveis, podem ser argumentos para funções e podem ser retornados como resultados.

Exemplo

```
a = 1
print(a) --> 1
a = "sou uma string"
print(a) --> sou uma string
a = true
print(a) --> true
```

Valores e Tipos de Dados em Lua

Há oito tipos de dados em Lua:

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

Valores e Tipos de Dados em Lua
Nil

- Nil tem apenas um valor, o **nil**, e sua principal característica é ser diferente de qualquer outro valor.
- Toda variável global tem valor o valor **nil** como padrão antes de sua primeira atribuição.

Valores e Tipos de Dados em Lua
Boolean

- Tipo booleano tradicional, com valores **true** e **false**.
- No entanto, o tipo Boolean não tem monopólio sobre valores de condicionais. Qualquer valor pode representar um valor condicional.

Definição

Em Lua, condicionais consideram **false** e **nil** como **falso** e quaisquer outros valores como **verdadeiro**.

Valores e Tipos de Dados em Lua
Number

- O tipo **Number** representa números reais (ponto-flutuante com dupla precisão).
- No desejo de representação de um inteiro, apenas para números maiores que 10^{14}, há erro de arredondamento (ou seja, Lua pode representar qualquer inteiro de até 32 bits sem erros).

Valores e Tipos de Dados em Lua
Number

- O tipo **Number** representa números reais (ponto-flutuante com dupla precisão).
- No desejo de representação de um inteiro, apenas para números maiores que 10^{14}, há erro de arredondamento (ou seja, Lua pode representar qualquer inteiro de até 32 bits sem erros).
- Lua 5.3 adicionou duas representações internas ao tipo **number**: *integer* e *float*.

Valores e Tipos de Dados em Lua
String

- Strings em Lua são sequências de caracteres.
- Strings em Lua são *8-bit clean*, o que significa que podem conter qualquer valor de 8 bits (por conseguinte, suporta UTF-8).
- São imutáveis. Não é possível mudar um caracter.
- É apenas possível criar uma nova string com as modificações desejadas.

Exemplo

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

Valores e Tipos de Dados em Lua
String

- Strings podem ser delimitadas com aspas simples ou duplas.
- Lua permite a conversão automática entre números e strings em tempo de execução. Qualquer operação numérica em uma string tenta convertê-la em um número.
- Lua aplica tais coerções sempre que espera um número como argumento.
- Para explicitar tais conversões, existe o operador de concatenação e as funções **tostring** e **tonumber**.

Exemplo

```
print(322 .. "" == "322") --> true
print(tostring(8001) == "8001") --> true
print(tonumber("420") == 420) --> true
```

- Funções são o principal mecanismo para abstração em Lua. Podem tanto executar procedimentos ou calcular e retornar valores.

Exemplo

```
--> executa procedimento
print(6*7)

--> retorna um valor real
a = math.sin(3) + math.cos(10)
```

- Sintaxe de uma função em Lua:

Exemplo

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

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

Exemplo

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

f(3)  --> a=3, b=nil
f(3, 4)  --> a=3, b=4
f(3, 4, 5)  --> a=3, b=4 (5 descartado)
```

- É possível retornar múltiplos valores de uma função.

Exemplo

```
function foo2 ()
  return "a","b" -- retorna 2 valores
end
x,y = foo2()
print(x .. " e " .. y) --> a e b
```

- Há suporte também para funções com número variado de argumentos.

Exemplo

```
function add (...)
  local s = 0
  for i, v in ipairs{...} do
    s = s + v
  end
  return s
end
print(add(3, 4, 10, 25, 12)) --> 54
```

- A passagem de parâmetros é posicional.
- O mecanismo de passagem para os tipos *nil*, *boolean*, *number* e *string* é **cópia**, enquanto para os tipos *function*, *userdata*, *thread* e *table* a passagem é por **referência**.
- Lua suporta funções aninhadas, e portanto, suporta closures:

Closures em Lua

```
function addto(x)
  return function(y) --funcao anonima
    return x + y
  end
end
fourplus = addto(4)
print(fourplus(3))  --> 7
```

- O tipo **userdata** permite que dados arbitrários em C sejam guardados em variáveis de Lua.
- Userdata não possui operações predefinidas, a não ser atribuição e teste de igualdade.
- São usados principalmente para representar novos tipos criados por uma aplicação ou uma biblioteca em C.
- O tipo **thread** representa linhas de execução independentes e são usadas para a implementação de corotinas. Lua provê suporte à corotinas até para sistemas que não suportam threads.

Valores e Tipos de Dados em Lua
Table

- O tipo *table* implementa arrays associativos.
- Não possuem tamanho fixo. É possível adicionar quantos elementos quisermos dinamicamente.
- Tables são o único mecanismo para implementação de estruturas de dados em Lua, e com isso, é poderoso o suficiente para a representação de vetores, matrizes, conjuntos, filas, listas, entre outras estruturas, de maneira simples, uniforme e eficiente.

Tables em Lua

```
I_am_Table = {}
```

Valores e Tipos de Dados em Lua
Table

- Assim como funções, tabelas são anônimas, ou seja, não há relacionamento fixo entre a variável que referencia a tabela e a tabela em si.
- Quando não há mais variáveis que referenciem uma tabela, o coletor de lixo eventualmente apaga a tabela.
- Pode-se acessar um item de uma tabela em Lua de duas maneiras:

Tables em Lua

```
tabela = {x = 5}
print(tabela["x"]) --> 5
print(tabela.x) --> 5
```

Cuidado!

Escrever `tabela[x]` é diferente de `tabela.x` e `tabela["x"]`. A primeira forma indica que um elemento da tabela é indexada pelo valor presente na variável `x` (a chave é o valor de `x`).

Exemplo

```
a = {}
x = "y"
a[x] = 10 -- atribui 10 ao campo "y"
print(a[x]) --> 10 -- valor do campo "y"
print(a.x) --> nil -- valor do campo "x"
print(a.y) --> 10 -- valor do campo "y"
```

Representação de um vetor convencional:

Exemplo

```
a = {}
for i=1,10 do
  a[i] = io.read()
end
-- imprimir os elementos
for i=1, #a do
  print(a[i])
end
```

Tables
Metatables e Metamethods

- Metatabelas permitem a mudança de comportamento de um valor em uma operação indefinida.
- Podemos assim, por exemplo, implementar a operação de soma entre duas tabelas.
- Tables e **userdata** tem metatabelas individuais, ao contrário dos outros tipos que possuem apenas uma metatabela para todos os seus valores.
- Lua permite apenas a manipulação de metatabelas somente de tabelas.

Exemplo

```
t = {}
print(getmetatable(t))     --> nil
t1 = {}
setmetatable(t, t1)
print(getmetatable(t)) --> table: 0x1423970
```

Tables
Metatables e Metamethods

- Metamétodos são mecanismos que permitem a sobrecarga de certas operações em objetos.
- Definem as funções que referem à comportamentos específicos de um objeto.
- O nome de um metamétodo é precedido por dois subtraços. Por exemplo, o metamétodo que representa a adição tem como nome **__add**.

Lista de metamétodos:

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

Tables
Orientação à Objetos em Lua

- Lua não oferece mecanismos explícitos para implementação de classes.
- No entanto, é possível simular tais mecanismos usando tabelas e funções.
- Herança pode ser implementada usando metatabelas.


```lua
Account = {}
Account.__index = Account

function Account.create(balance)
    local acnt = {}  -- o novo objeto
    --seta Account como metatable de acnt
    setmetatable(acnt,Account)
    -- inicializacao do objeto
    acnt.balance = balance
    return acnt
end

function Account:withdraw(amount)
    self.balance = self.balance - amount
end
-- criando e usando uma conta
acc = Account.create(1000)
acc:withdraw(100)
```

- Lua não provê mecanismos para criar constantes na linguagem.
- Lua usa coletor de lixo desde sua criação. Até a versão 5.0, a estratégia usada era o **marcar-varrer**.
- A partir da versão 5.1, Lua passou a ter um coletor marcar-varrer **incremental**.

- Lua não provê mecanismos para serialização na linguagem.
- No entanto, a implementação não é considerada complicada.
- `http://lua-users.org/wiki/TableSerialization` provê várias funções feitas pela comunidade de Lua para serialização de tabelas.

- Mecanismos para criação de módulos e pacotes (coleção de módulos) em Lua foram criados somente a partir da versão 5.1, com a introdução das funções **require**, para usar módulos e **module**, para criar módulos.
- Lua 5.2 removeu a função **module**, mantendo apenas **require**.

```
--importa o modulo Autor
require "Classes.Autor"

--importa o modulo io e permite
-- a chamada de funcoes do modulo usando
-- rev.funcao()
local rev = require "io"
rev.write("Yeh I'm a wrathchild.\n")
```

- Lua suporta os seguintes tipos de polimorfismo: **coerção**, **sobrecarga** e **inclusão**.

- Com a adição dos subtipos *integer* e *float* em Lua 5.3, a operação de amplicação é segura, nunca falhando. A operação de estreitamento, no entanto, só funciona caso o *float* em questão tiver uma representação exata nos inteiros (ex.: 10.0).
- Há também as conversões de *string* para *number*, sempre que uma operação numérica está para ser realizada e uma *string* com representação exata de um número é usada.

Coerção em Lua

```
x = 7 + 8.8 --> 15.8 (Lua 5.3)
y = "10" + 12 --> 22
z = "17" + "71" --> 88
```

Polimorfismo
Sobrecarga

- Lua não suporta sobrecarga de funções, mas é possível implementar com base nos argumentos que a função recebe.
- Operadores são sobrecarregados com o uso de metamétodos.

```
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

Apesar de Lua não suportar classes explicitamente, o mecanismo de herança pode ser simulado, também com tabelas e metatabelas.

```
--Fazer ContaCorrente herdar de Account
ContaCorrente = Account.create()
--criacao de metatabela de ContaCorrente
ContaCorrente.__index = ContaCorrente

function ContaCorrente.create(numero,balance)
  local contacorrente = Account.create(balance)
  setmetatable(contacorrente,ContaCorrente)
  --variavel de ContaCorrente
  contacorrente.numero = numero

  return contacorrente
end
--metodo de ContaCorrente
function ContaCorrente:deposit(qtd)
  self.balance = self.balance + qtd
end

rtz = ContaCorrente.create(10931105,400)
rtz:deposit(105)
print(rtz.balance) --> 505
```

Exceções
Exceções e tratamentos de erro em Lua

- Usualmente, erros encontrados durante tempo de execução causam o fim do programa.
- O tratamento de erros é feito pelas funções **pcall** e **error**.

```
function C()
  print("C 1")
  print(1 + nil)
  print("C 2")
end
--faz uma chamada protegida a funcao C,
--capturando erros que a funcao lance
code,erro = pcall(C)
print(code and "Success" or erro)
-->C 1
-->bots.lua:16: attempt to perform
-->arithmetic on a nil value
```

Uso da função *error*:

```
status, err = pcall(function ()
                    error("my error",1) end)
print(err) -->bots.lua:31: my error
```

- Lua não provê suporte à **threads**, mas oferece um mecanismo similar: **corotinas**.
- No entanto, enquanto em uma máquina com múltiplos processadores, temos várias threads de um programa rodando simultaneamente, corotinas são colaborativas, ou seja, somente uma corotina executa por vez, e só suspende sua execução caso requisite explicitamente sua suspensão.
- As funções relacionadas à corotinas são armazenadas na tabela de corotinas. A função **create** cria novas corotinas, tendo como único argumento a função com o código que a corotina executará. Retorna um valor do tipo **thread**, que representa a corotina.

Corotinas

```
function foo()
    for i=1,10 do
      print("oi, eu sou o Goku")
   end
   coroutine.yield() --> suspende a corotina
end

--cria a corotina
co = coroutine.create(foo)
print(coroutine.status(co)) --> suspended
coroutine.resume(co)
print(coroutine.status(co)) --> suspended
```

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**

Sim

Lua é descrita como uma linguagem de extensão de propósito geral.

- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**

Parcial

Lua possui um mecanismo para tratamento de erros, além de coletor de lixo, mas possui mecanismos que podem induzir a erros, como desvio incondicional irrestrito.

- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**

Sim

Lua possui uma sintaxe simples e limpa, sendo bem legível. O uso de tabelas requer prática, mas nada muito complicado.

- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**

Sim

Lua é conhecida como uma das mais rápidas linguagens de script, além de ser leve e usar pouca memória.

- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**

Sim

Por ter como a facilidade de integração como característica fundamental, a portabilidade também se torna outra característica importante para a linguagem.

- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**

Estruturado, OO (baseado em protótipos)

- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**

Sim

- **Reusabilidade**
- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**

Sim

Apesar de maneira diferente de outras linguagens, Lua fornece mecanismos para reuso, como módulos, além de suportar polimorfismo universal.

- **Integração**
- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**

Sim

Lua foi designada para ser usada junto com C. Mas também pode ser integrada com Java, Python, Ruby, dentre outras linguagens.

- **Custo**

Avaliação da linguagem
Critérios gerais

- **Aplicabilidade**
- **Confiabilidade**
- **Aprendizado**
- **Eficiência**
- **Portabilidade**
- **Método de Projeto**
- **Evolutibilidade**
- **Reusabilidade**
- **Integração**
- **Custo**

Depende da ferramenta

- **Escopo**

Parcial

A não definição de entidades em Lua significa que possuem o valor *nil*. Porém, toda definição associa uma entidade a um escopo de visibilidade, sendo como padrão o escopo **global**.

- **Expressões e comandos**
- **Tipos primitivos e compostos**
- **Gerenciamento de memória**
- **Persistência dos dados**
- **Passagem dos parâmetros**

Avaliação da linguagem
Critérios específicos

- **Escopo**
- **Expressões e comandos**

Sim

- **Tipos primitivos e compostos**
- **Gerenciamento de memória**
- **Persistência dos dados**
- **Passagem dos parâmetros**

Avaliação da linguagem
Critérios específicos

- **Escopo**
- **Expressões e comandos**
- **Tipos primitivos e compostos**

Parcial

Lua fornece apenas oito tipos, mas com o tipo tabela, é possível implementar muitas estruturas e tipos abstratos de dados.

- **Gerenciamento de memória**
- **Persistência dos dados**
- **Passagem dos parâmetros**

- **Escopo**
- **Expressões e comandos**
- **Tipos primitivos e compostos**
- **Gerenciamento de memória**

Sistema

- **Persistência dos dados**
- **Passagem dos parâmetros**

- **Escopo**
- **Expressões e comandos**
- **Tipos primitivos e compostos**
- **Gerenciamento de memória**
- **Persistência dos dados**

Biblioteca de funções e serialização

Lua oferece algumas funções para I/O, enquanto interface com banco de dados e serialização são fornecidos por implementações feitas por usuários.

- **Passagem dos parâmetros**

Avaliação da linguagem
Critérios específicos

- **Escopo**
- **Expressões e comandos**
- **Tipos primitivos e compostos**
- **Gerenciamento de memória**
- **Persistência dos dados**
- **Passagem dos parâmetros**

Lista variável, default, por valor e por referência

- **Encapsulamento e proteção**

Parcial

- **Sistema de tipos**
- **Verificação de tipos**
- **Polimorfismo**
- **Exceções**
- **Concorrência**

Avaliação da linguagem
Critérios específicos

- **Encapsulamento e proteção**
- **Sistema de tipos**

Sim

Lua é bastante rígido quanto à violações no sistema de tipos, incluindo proibir coerção de estreitamento.

- **Verificação de tipos**
- **Polimorfismo**
- **Exceções**
- **Concorrência**

- **Encapsulamento e proteção**
- **Sistema de tipos**
- **Verificação de tipos**

Dinâmica

- **Polimorfismo**
- **Exceções**
- **Concorrência**

- **Encapsulamento e proteção**
- **Sistema de tipos**
- **Verificação de tipos**
- **Polimorfismo**

Coerção, sobrecarga e inclusão

- **Exceções**
- **Concorrência**

Avaliação da linguagem
Critérios específicos

- **Encapsulamento e proteção**
- **Sistema de tipos**
- **Verificação de tipos**
- **Polimorfismo**
- **Exceções**

Parcial

Lua fornece mecanismos para tratamento de erros, mas não obriga seu uso.

- **Concorrência**

Avaliação da linguagem
Critérios específicos

- **Encapsulamento e proteção**
- **Sistema de tipos**
- **Verificação de tipos**
- **Polimorfismo**
- **Exceções**
- **Concorrência**

Parcial

Provê o mecanismo de corotinas, que são diferentes de *threads*. Não provê recursos para exclusão mútua.

Referências

- *Programming in Lua*, 2006, por Roberto Ierusalimschy
- *Beginning Lua Programming*, 2007, por Kurt Jung e Aaron Brown
- `http://www.lua.org/manual/5.3/`
- `http://www.luafaq.org/`
- `http://lua-users.org/wiki/`