ANDRÉ LUIZ PIONA DE ARAÚJO
PEDRO PAIVA LADEIRA

Introdução

Introdução

Lua é uma linguagem de origem brasileira, que teve como foco inicial, ser integrado com softwares escritos em C e outras linguagens convencionais. Isso trouxe como benefício o fato dele ser pequeno e simples.

Ele não teve como objetivo fazer o que C já faz bem, e sim o que C não é bom, como a distância do hardware na hora de programar, estruturas dinâmicas e facilidade para testar e debugar.

História

Lua foi criada em 1993 por Tecgraf da PUC-Rio, para ser utilizada em um projeto da Petrobrás. Devido ao sucesso pela sua eficiência e facilidade, passou a ser usado em vários ramos da programação, como criação de jogos, controle de robôs, etc.

Hoje Lua está na versão 5.3, e é licenciado pela MIT desde a versão 5.0. É uma das principais linguagens usadas para script.

Características

- Software livre de código aberto;
- Linguagem mais rápida entre as de script interpretadas;
- Compila em qualquer plataforma com compilador C;
- Tipada dinamicamente;
- Case-sensitive;
- Interpretada a partir de bytecodes;
- É multi-paradigma;

Começando

Para começar e testar funções básicas é possível no próprio site da linguagem (www.lua.org) na sessão Getting Started há um live demo em que o usuário pode testar as funções mais básicas. Há também alguns modelos de programas básicos.

Começando(2)

Caso queira começar a programar em Lua no próprio computador basta baixar na sessão de downloads do site da linguagem.

Para programar em Lua é necessário um compilador de C (GCC por exepmlo) e baixar o pacote fonte do Lua. No windows, além disso, é necessário utilizar alguns scripts em shell para configurar o ambiente da linguagem Lua (http://lua-users.org/wiki/BuildingLuaInWindowsForNewbies).

Além disso há uma IDE para LUA, a ZeroBrane Studio (https://studio.zerobrane.com/)

“Hello World”

Para utilizar a linguagem Lua, basta digitar seu código em um arquivo .lua, e executar na prompt usando “lua <arquivo.lua>”.

```
--hello.lua
print (“Hello World”)
--end of file


prompt> lua hello.lua          → Hello World
```

Modo Interativo

Lua também pode rodar em modo interpretador stand-alone, chamado de “Modo Interativo”. Para utilizá-lo basta digitar “lua” na prompt.

Neste modo, se digitar um comando e apertar enter, o comando será executado imediatamente, mas é possível usar funções de um arquivo .lua usando o comando *dofile*.

```
--arquivo ‘lib1.lua’
function norm (x, y)
    local n2 = x^2 + y^2
    return math.sqrt(n2)
end
```

```
Lua 5.3.4 Copyright (C) 1994-2017
Tecgraf, PUC-Rio
> dofile("lib1.lua")
> n = norm(3, 7)
> print(n)    → 7.6157731058639
```

Organização

Não é necessário separar comandos consecutivos, porém é recomendado usar ponto e vírgula quando tiver mais de um comando em uma mesma linha.

```
a = 1
b = a*2
```

```
a = 1; b = a*2
```

```
a = 1;
b = a*2;
```

```
a = 1 b = a*2
```

Todos são válidos.

Cuidado

Em Lua você deve evitar usar identificadores começando com underline seguido com uma ou mais letras maiúsculas (exemplo `_VERSAO`), eles são reservados para usos especiais.

É possível utilizar caracteres especiais para nome de variáveis, porém não é recomendado também.

Palavras Reservadas

As seguintes palavras são reservadas e não podem ser usadas como identificadores:

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

Comentando

Para comentar uma parte do código, é possível usar dois hífens seguidos (`--`) no começo caso seja apenas uma linha, ou usar '`--[[`' antes da parte do código, e '`]]--`' depois.

```
-- print(10)

--[[
a = 3
b = a*2
]]--
```

Valores e Tipos

Valores e Tipos

Como lua é uma linguagem dinamicamente tipada, o usuário não define o tipo da variável, o tipo é determinado automaticamente dependendo do valor atribuído.

Lua possui oito tipos básicos de variáveis: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread* e *table*.

Existe uma função chamada *type*, ela retorna o tipo da variável.

```
print(type("Hello world"))      → string
```

Nil

Nil é um tipo que possui apenas o valor **nil**. Ele tem como objetivo representar a falta de qualquer valor significante na variável.

Qualquer variável terá o valor **nil** como padrão, antes de receber outro valor. Ele pode ser atribuído para deletar uma variável.

Boolean

O tipo boolean possui apenas dois valores, **true** e **false**, para representar os valores tradicionais boolean.

Em Lua, as estruturas condicionais visualizam como false apenas os valores **false** e **nil**, todo resto é considerado **true**. Ou seja, diferente do costume seguido nas outras linguagens, tanto o zero quanto uma string vazia são considerados **true**.

Números

O tipo numérico representa tanto integer, como números reais (float).

Exemplo de constantes numéricas aceitas:

- 4
- 0.4
- 4.57e-3
- 0.3e12

Strings

String em Lua são valores imutáveis. Não é possível alterar uma letra dentro de uma string, por exemplo. Para alterar algum valor é necessário criar uma nova string.

```
a = “uma string”
b = string.gsub(a, “uma”, “alguma”)
print (b)                    →  alguma string
```

Userdata

O tipo userdata permite que dados arbitrários em C sejam guardados em variáveis de Lua.

Lua não possui operações pré-definidas para manipular o tipo userdata, exceto para atribuir e identificar o tipo. O restante deve ser feita através de um API de C.

Function

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

Em Lua, uma função pode retornar múltiplos valores:

```
function soma (a,b)
    return a, b, a+b
end

a,b = 1,2
print(soma(a,b))      → 1   2   3
```

Function

Para chamar uma função, não é necessário passar uma quantidade exata de parâmetros. Quando é passado um número inferior de parâmetros, são atribuídos nil para os valores restantes, e em caso de valores superiores de parâmetros, os extras são descartados.

```
function teste (a,b)
    return a+b
end

a,b,c = 1,2,3
print(soma(a,b,c))          → 3          #a=1, b=2
print(soma(a))              → Erro       #a=1, b=nil
```

Function

As passagens dos tipos *nil*, *boolean*, *numérico* e *string* são de **cópia**, enquanto os tipos *function*, *userdata*, *thread* e *table* são de **referência**.

Lua suporta funções aninhadas, e com isso, closures também.

```
function teste (a)
     return function (b)
          return (a+b)
     end
end

x = teste(2)
print (x(3))        → 5
```

Thread

O tipo thread representa linhas de execução independentes e são usadas para a implementação de co-rotinas.

Threads do Lua não são relacionadas com as threads do sistema operacional.

Lua suporta co-rotinas em todos os sistemas, até os que não possuem suporte nativo a threads.

Table

O tipo ***table*** implementa arrays associativos, isto é, arrays que podem ser indexados não apenas com números, mas com qualquer valor Lua, exceto **nil**.

Tabelas (table) são o único mecanismo de estruturação de dados em Lua. Elas podem ser usadas para representar arrays comuns, sequências, tabelas de símbolos, conjuntos, registros, grafos, árvores, etc.

Para criar uma tabela vazia:

```
novaTabela = {}
```

Table

A diferentes formas de acessar um item em uma tabela, exemplo:

```
novaTabela = {x = 3}
print (novaTabela["x"])        → 3
print (novaTabela.x)           → 3
print (novaTabela[x])          → nil
y = "z"
novaTabela[y] = 4
print (novaTabela[y])          → 4
print (novaTabela["z"])        → 4
print (novaTabela.z)           → 4
print (novaTabela.y)           → nil
```

Escopo

O escopo em Lua é um escopo léxico.

Para criar uma variável para ser usado apenas em uma função, é necessário declarar ela como `local`.

Por conta das regras de escopo léxico, variáveis locais podem ser livremente 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.

Escopo (Exemplo)

```
x = 10                  # variável global
 do
   local x = x          # variável local
   print(x)             →  10
   x = x+1
   do
     local x = x+1      # variável local de função interna
     print(x)           →  12
   end
   print(x)             →  11
 end
 print(x)               →  10
```

Coletor de Lixo

Lua gerencia sua memória automaticamente, ou seja, não é necessário se preocupar em alocar ou liberar memória para os objetos. Esse gerenciamento vale para todos os tipos (strings, tables, userdata, etc.).

É utilizado um coletor *mark-and-sweep* incremental. Ele usa dois números para controlar seus ciclos: de pausa para controlar o tempo entre um ciclo e outro, e de multiplicador de passo para controlar a velocidade que é realizado o ciclo.

É possível alterar esses dois valores através de seus *metamétodos*.

Expressões

Expressões

As expressões em Lua incluem constantes numéricas e strings literais, variáveis, operações unárias e binárias e chamadas de funções. Além disso incluem definições de funções e construtores de tabelas

Operadores Aritméticos

Lua suporta os operadores aritméticos básicos que já conhecemos:

Soma	Subtração ou Negação	Divisão	Multiplicação	Modulo	Exponencial
+	–	/	*	%	^

Estes operadores funcionam para números reais ou para tipos que possam ser interpretados ou convertidos para números reais.

Operadores Relacionais

Os operadores relacionais que o Lua suporta são os seguintes:

Maior que	Menor que	Maior ou igual	Menor ou igual	Idêntico	Não idêntico
>	<	>=	<=	==	~=

Estes operadores sempre retornam como resultado um booleano. Os tipos ‘table’ e ‘userdata’, nestes casos, são comparados por referência, ou seja, só são iguais se forem literalmente o mesmo objeto.

Operadores Lógicos

Lua tem como operadores lógicos **or**(OU), **and**(E) e **not**(NÃO).

O operador **not** sempre retorna um booleano.

O operador **and** retorna o primeiro argumento se o mesmo for **falso**, caso contrário retorna o segundo.

O operador **or** retorna o primeiro argumento se o mesmo for **não falso**, caso contrário retorna o segundo.

As operações **or** e **and** são operações consideradas de curto-circuito.

Operadores Lógicos

`print(1 and 2)`	`2`
`print(nil and 2)`	`nil`
`print(false and 2)`	`false`
`print(1 or 2)`	`1`
`print(false or 2)`	`2`
`print(not false)`	`true`
`print(not 0)`	`false`
`print(not 15)`	`false`
`print(not not nil)`	`false`

Operador de Concatenação

O operador de concatenação é dado por: ‘ .. ‘

Funciona para tipos **String**, porém se utilizado com um **Number** concatena interpretando o Number em questão como uma String.

`print(“C” .. 3 .. “P” .. 0)`	`C3P0`
`print(“Agente ” .. “Secreto ” .. 007)`	`Agente Secreto 7`
`print(“Agente ” .. “Secreto ” .. 0 .. 0 .. 7)`	`Agente Secreto 007`
`print(0101 .. “Dálmatas”)`	`101Dálmatas`

O Operador Length

O operador length é representado por `#`

Funciona para tables e strings, no caso das strings retorna o número de bytes da string, para tabelas retorna o tamanho da sequência representada pela tabela.

Para strings esse operador é eficiente, mas para tabelas quando possui buracos a resposta pode ser imprevisível.

O Operador Length(2)

a=”Galinha” print(#a)	7
a=”Quiabo” print(#a)	6
a={} a[1]=1 a[2]=1 a[3]=nil a[4]=1 print(#a)	4
a={} a[1]=1 a[2]=1 a[3]=1 a[4]=nil a[5]=1 print(#a)	3

Precedência

A ordem de prioridade dos operadores que vimos até agora segue pela seguinte tabela:

1	^
2	not # - (unário)
3	* / %
4	+ -
5	..
6	< > <= >= ~= ==
7	and
8	or

Quando em dúvida é sempre recomendável usar parênteses para explicitar as intenções de prioridade nas expressões.

Construtor de Tabelas

O construtor mais simples é o de tabela vazia

```
{}
```

Construtores podem também inicializar listas:

```
Reservoir = {“Blonde”,”Orange”, “Pink”, “Brown”,
“White”,”Blue”}
```

```
print(Reservoir[3]) →  Pink
```

O primeiro índice das listas é 1 e não 0.

Construtor de Tabelas(2)

O construtor de tabelas do Lua também cria tabelas ‘record-like’:

```
a={x=3,y=2}
```

```
a={}; a.x=3; a.y=2        -- Este comando é equivalente ao da linha acima
```

As tabelas são extremamente flexíveis, ou seja, independente do construtor inicialmente usado elas podem ter campos adicionados e removidos, mas é sempre mais eficiente e elegante quando o construtor correto é utilizado.

Construtor de Tabelas(3)

Também é possível criar com construtores tabelas meio lista meio ‘record-like’

```
a{x=10, y=45; "um", "dois", "três"}  -- o “;” poderia ser “,” e
                                     -- não alteraria a tabela

print(a.x)
print(a.y)
print(a[1])
print(a[2])
print(a[3])
```

```
10
45
um
dois
três
```

Estruturas de Controle

As estruturas de controle que o Lua suporta são:

`if/else`	`while`	`repeat/until`	`for numérico`	`for genérico`

If/Else

Estrutura de controle básica da maioria das linguagens.

```
if(condição) then
    ...
end
```

```
if(condição) then
    ...
else
    ...
end
```

```
if(condição1) then
    ...
elseif(condição2) then
    ...
elseif(condiçãoN) then
    ...
else
    ...
end
```

While

No while uma condição é conferida e a ação é repetida até a condição parar de ser verdadeira.

```
while(condição)do
    …
end
```

Repeat/Until

Funciona de forma parecida com while, porém o bloco de código roda pelo menos uma vez, pois a condição é checada depois da execução do mesmo:

`i=5`
`repeat`
`  i=i-1`
`  print(i)`
`until(i==0)`
	`4`
	`3`
	`2`
	`1`
	`0`

`i=5`
`repeat`
`  i=i-1`
`  print(i)`
`until(i<6)`
	`4`

For Numérico

A sintaxe do for numérico é a seguinte:

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

Onde os valores de exp1, exp2 e exp3 são sempre numéricos. exp1 representa o valor inicial, exp2 representa o valor final e exp3 representa o passo que o valor da variável de controle dará para chegar no valor final. Por default o passo implícito é somar 1.

```
--Passo implícito
for i = 1,15 do

    ...  --Incrementará i de 1 em 1
         --por default
end
```

```
--Passo explícito
for i = 15,1,-1 do

    ...  --Decrementará i de 1 por vez,
         --pois foi explicitado na exp3
end
```

For Genérico

Funciona percorrendo valores por uma função iteradora. Existem algumas funções iteradoras fornecidas pelas bibliotecas básicas, mas você pode fazer uma customizada. De qualquer forma exemplificaremos utilizando **pairs**, uma função iteradora para percorrer tabelas. A sintaxe de um for genérico é a seguinte:

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

Para cada passo do loop k pega uma chave enquanto v pega um valor relacionado a essa chave.

Estruturas de Controle

Lua suporta goto, break e return:

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

```
a = {1,2,3,4}
for v, i in pairs (a) do
      if v == 3 then
         break
      else
         print(v)
      end
end
```

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

Orientação à Objetos

Orientação à Objetos em Lua

Lua não possui explicitamente a orientação a objetos, mas é possível implementar com a ajuda de tabelas e funções de primeira classe de Lua. Ao colocar funções e dados relacionados em uma tabela, um objeto é formado. A herança pode ser implementado com a ajuda de metatabelas.

Metatabela (Metatable)

Em Lua, cada operação que um valor pode fazer (somar, diminuir, imprimir, etc.) é chamado de metamétodo (metamethod), e uma tabela de metamétodos é chamada de metatabela (metatable).

Todo valor pode ter uma metatabela, mas só tabela e userdata podem ter metatabelas individuais. Todos os valores de outros tipos dividem uma mesma tabela para aquele tipo.

Criando Metatabela

Uma metatabela é inicialmente definido da mesma forma que uma tabela, porém é utilizado o comando `setmetatable(tabela,metatabela)` para definir uma tabela como metatabela da outra.

```
tabela = {}
metatabela = {}
setmetatable(tabela, metatabela)
```

Criando Metamétodo

Para criar um metamétodo para uma tabela, tem que usar dois underlines seguidos da operação que deseja alterar, por exemplo:

```
tabela = setmetatable({chave1 = "valor1"}, {
   __index = function(tabela, chave)
      if chave == "chave2" then
         return "valorMetatabela"
      end
   end
})

print(tabela.chave1,tabela.chave2)              → valor1
valorMetatabela
```

Alguns Metamétodos

`__add`	Altera o comportamento do operador ‘+’.
`__sub`	Altera o comportamento do operador ‘-’.
`__mul`	Altera o comportamento do operador ‘*’.
`__div`	Altera o comportamento do operador ‘/’.
`__mod`	Altera o comportamento do operador ‘%’.
`__concat`	Altera o comportamento do operador ‘..’.
`__eq`	Altera o comportamento do operador ‘==’.
`__lt`	Altera o comportamento do operador ‘<’.
`__le`	Altera o comportamento do operador ‘<=’.

Criando uma Classe

Para criar uma classe, é necessário criar uma metatabela que será usada dentro do construtor como base para os objetos que serão criados.

```
Conta = {}
Conta .__index = Conta

function Conta:novo (saldo)
   local novaConta = {}            -- Cria novo objeto
   setmetatable(novaConta,Conta)   --  Seleciona a metatable da classe
   novaConta.saldo = saldo         -- Inicializa o objeto
   return novaConta
end
```

Criando um Novo Método

Após criar o construtor e a metatable da classe, criar um método novo para a classe é simples, basta criar uma função com o nome da classe seguido de “:” e o nome que deseja do método.

```
function Conta:saque(quantia)
   self.saldo = self.saldo - quantia
end

contaTeste = Conta:novo (1000)
contaTeste:saque(100)
print(contaTeste.saldo)
```

Herança

Para atribuir uma herança, é necessário criar um objeto da classe que deseja herdar, e atribuir um novo metatable para ele. É possível substituir um método da classe herdada, criando um método de mesmo nome usando a nova classe.

```
ContaEspecial = Conta:nova()
ContaEspecial.__index = ContaEspecial

function ContaEspecial:new(saldo)
     novaCEspecial = Conta:novo(saldo)
     setmetatable(novaCEspecial, ContaEspecial)
     novaCEspecial.juros = 0.05
     return novaCEspecial
end
```

Modularização

Modularização

Lua é uma linguagem que costuma não definir padrões a serem seguidos e sim formas de criar os próprios padrões de acordo com a necessidade. Para modularização não é tão simples, é necessária a padronização do sistema de módulos para facilitar o compartilhamento de conteúdo.

Na versão 5.1 foi definido o sistema de módulos ao inserirem as funções **module** e **require**. Na versão 5.2 a função **module** foi retirada e apenas a **require** se manteve.

Função Require

A função **require** carrega um módulo, módulos são, de certa forma, tabelas que contém funções e conteúdos necessários para o funcionamento do módulo.

```
require “modname”
```

Require checa na tabela **package.loaded** se o módulo já está carregado. Se não ele procura um arquivo Lua com o nome “modname”, e quando acha o carrega.

Para utilizar um módulo novo basta colocá-lo numa pasta onde a função require o encontrará.

Bibliotecas Padrão

Não é necessário carregá-las utilizando a função **require**.

Fornecem funções úteis, serviços essenciais (type, getmetatable) e serviços externos(I/O).

As bibliotecas padrão são implementadas pela API de C e são fornecidos como módulos separados em C.

Bibliotecas Padrão(2)

As bibliotecas padrão de Lua são as seguintes:

basic library
coroutine library
package library
string manipulation
basic UTF-8 support
table manipulation
mathematical functions
input and output
operating system facilities
debug facilities

Polimorfismo

Polimorfismo

Lua suporta os seguintes tipos de polimorfismo:

- Coerção
- Sobrecarga
- Inclusão

Coerção

Como citado no **operador de concatenação** quando utilizado tipo Number nesse tipo de operação o Number é convertido em String.

```
print(“blink” .. 182)  --> blink182
```

Quando uma String é completamente numérica e ela é utilizada numa operação aritmética ela é convertida para Number.

```
print(12*"2") --> 24
```

```
print(“blink”+182) --> erro!
```

Sobrecarga

Sobrecarga de operadores é feita por **Metamethods**.

Não há sobrecarga de funções.

Inclusão

Lua não é uma linguagem OO, mas ela provê mecanismos para simular classes como objetos. Por serem objetos as classes conseguem pegar métodos de outras classes. Nesse contexto é possível utilizar herança simples e múltiplas em classes.

Excessões

Tratamento de Erros e Exceções

Normalmente não é necessário tratar erros no Lua, todas as atividades começam por uma chamada que pede ao Lua para analisar um bloco. Se há um erro a própria chamada retorna o erro e continua a aplicação.

De qualquer forma o Lua oferece formas de tratar erros caso seja necessário, trata-se das funções **pcall** e **error**.

Função *error*

A função error chama um erro compulsoriamente no meio do código e imprime uma mensagem na tela:

```
print "enter a number:"
n = io.read("*n")
if not n then error("invalid input") end
```

Função *pcall*

A função pcall roda determinada função em modo protegido e captura determinado erro enquanto a função está sendo executada. Se não há erros pcall retorna true, do contrário retorna false mais a mensagem de erro. Normalmente são utilizadas funções anônimas, mas não necessariamente:

```
if pcall(function() … end) then …
    else …

local status, err = pcall(function() error({code=121})
end)

print(err.code)
```

```
--nenhum erro ocorreu
--tratamento de erro



-->121
```

Concorrência

Concorrência

Lua não suporta multithreading, mas suporta co-rotinas, que se assemelham muito a threads no sentido de que há uma linha de execução com sua própria pilha de execução, variáveis locais e contador de programa, mas dividem variáveis globais e o resto com outras co-rotinas.

Diferente das threads as co-rotinas operam uma de cada vez e, diferente de funções as co-rotinas podem ser suspensas e retomadas posteriormente de onde pararam.

Co-rotinas

As funções relacionadas a co-rotinas estão localizadas na tabela ***coroutine***. A função **create**, cria uma nova co-rotina que retorna o tipo thread.

```
co = coroutine.create (function () print("ola") end) -- a variável co é
                                                     -- tipo thread
```

Logo após sua criação a co-rotina fica em estado *Suspended*. Os 3 estados possíveis de uma co-rotina são *Suspended*, *Running* e *Dead*. Para saber o estado da co-rotina usamos a função **status**

```
print(coroutine.status(co)) -->   Suspended
```

Co-rotinas

Outra função interessante aplicada a co-rotinas é a função **resume** que faz com que uma co-rotina que está no estado *Suspended* vá para o estado *Running*.

A função yield suspende a co-rotina e retorna o controle para o resume. E é aí que as co-rotinas mostram sua diferença para uma simples chamada de função.

Co-rotinas

```
co = coroutine.create(function ()
        for i = 1, 10 do
            print("co", i)
            coroutine.yield()
        end
    end)
coroutine.resume(co) --> co 1
print(coroutine.status(co)) --> suspended
coroutine.resume(co) --> co 2
print(coroutine.status(co)) --> suspended
…
coroutine.resume(co) --> co 10
print(coroutine.status(co)) --> suspended
coroutine.resume(co) --> Não imprime nada, o loop chegou ao fim
coroutine.resume(co) --> false cannot resume a dead coroutine
```

Avaliação

Avaliação

Critérios gerais	C	C++	Java	Lua
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Parcial
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de Projeto	Estruturado	Estruturado e OO	OO	Estruturado e OO
Evolutibilidade	Não	Parcial	Sim	Sim

Avaliação

Critérios gerais	C	C++	Java	Lua
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e compostos	Sim	Sim	Sim	Parcial
Gerenciamento de memória	Programador	Programador	Sistema	Sistema

Avaliação

Critérios gerais	C	C++	Java	Lua
Persistência de dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Biblioteca de funções e serialização
Passagem de parâmetros	Lista variável por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, por cópia e por referência
Encapsulamento e proteção	Parcial	Sim	Sim	Parcial
Sistema de tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática/Dinâmica	Estática/Dinâmica	Dinâmica

Avaliação

Critérios gerais	**C**	**C++**	**Java**	**Lua**
Polimorfismo	Coersão e Sobrecarga	Todos	Todos	Coerção/ Sobrecarga/ Inclusão
Exceções	Não	Parcial	Sim	Parcial
Concorrência	Não (Biblioteca de funções)	Não (Biblioteca de funções)	Sim	Parcial

Referências

- https://www.lua.org
- http://lua-users.org/
- http://www.w3ii.com/pt/lua/default.html
- IERUSALIMSCHY, R. Programming in Lua. Third Edition. Rio de Janeiro. PUC-RJ, 2013.
- DE FIGUEIREDO, L.H; CELES, W.; IERUSALIMSCHY, R. Lua Programming Gems. Rio de Janeiro. PUC-RJ, 2008.