LUA

Carlos Henrique Maulaz de Freitas
Matheus Beloti Mariani
Thiago Gozzi Renoldi Siqueira Costa

1 O que é Lua?

O que é Lua?

Linguagem de programação de extensão projetada para dar suporte à programação procedimental.

Oferece suporte também para:

- Programação Orientada a Objetos
- Programação funcional
- Programação orientada a dados

Planejada para ser leve e eficiente.

HISTÓRIA

História

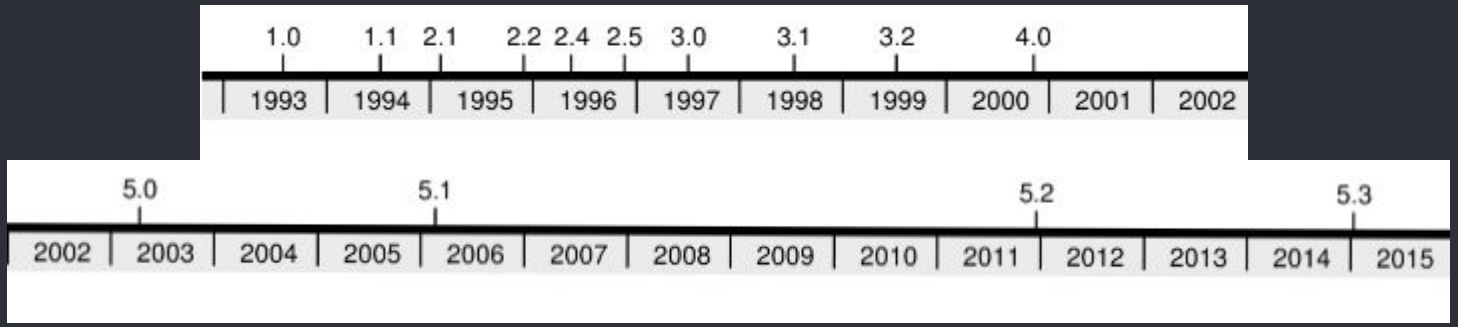
- Criada em 1993 por Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes

- Desenvolvida pela PUC-RJ

História

- Criada para ser utilizada em um projeto da Petrobrás
- Reconhecimento mundial
- Licença BSD
- A partir da versão 5.0, licença MIT.

História



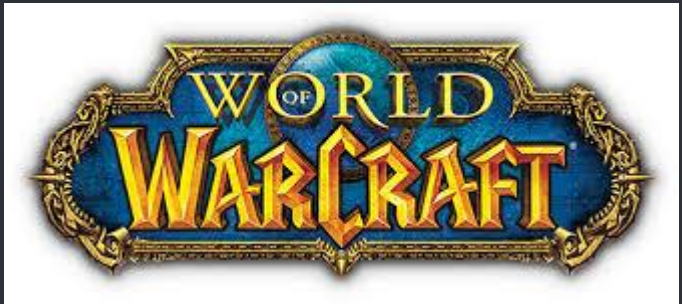
- 1994 - Primeira versão lançada para o público
- 1997 - Se descobre a aplicação para jogos
- 2002 - Licenciada pela MIT
- 2010 - Adobe começa a patrocinar

História

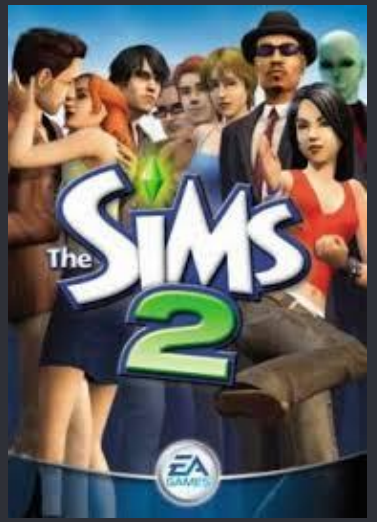
Ganhou bastante visibilidade após ser usada no jogo Grim Fandango, produzido pela LucasArts

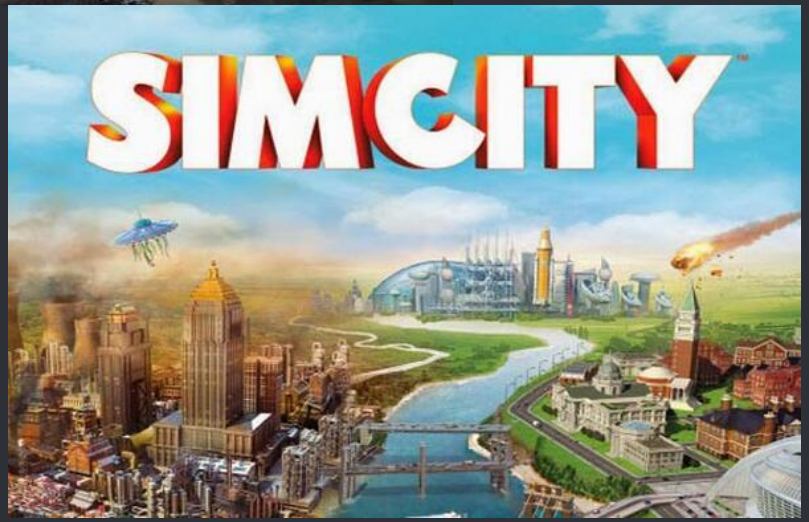
História

Utilizada em vários outros jogos:



História





Outras aplicações:

- Desenvolvimento de jogos
- Controle de robôs
- Processamento de textos
- Intercomunicação de base de dados em sites dinâmicos
- Entre outros

Outras aplicações:

Porque usar Lua?

Lua é uma linguagem estabelecida e robusta, com aplicações Industriais (Adobe), sistemas embutidos (Ginga) e é atualmente a linguagem script mais utilizadas em jogos.

Lua é rápida, vários benchmarks mostram que Lua é a linguagem mais rápida dentre as linguagem de script interpretadas.

- LuaJIT

Como obter?

www.lua.org

about
news
get started
download
documentation
community
contact
site map
português

Lua 5.3.3
released

Fourth edition of
Programming in Lua
published

Lua Workshop 2016
held in San Francisco

PUC
RIO

Características Gerais:

- Linguagem multi-paradigma
- Sintaxe simples e intuitiva
- Dinamicamente tipada
- Case-sensitive
- Linguagem interativa
- Código aberto

Execução

- Interpretada a partir de bytecodes em um máquina virtual
- Gerenciamento de memória automático com coleta de lixo incremental
- Pode se obter maior velocidade utilizando LuaJIT

TIPOS DE DADOS

Tipos de Dados

- Lua por ser uma linguagem dinamicamente tipada nela não existe definição de tipo, cada valor carrega seu próprio tipo.
- Todos os valores são de primeira classe.

Nil, boolean, number, string, table, function, userdata, thread.

Tipos de dados

função type(), retorna o tipo de um valor qualquer.

print(type(231.2346))

print(type(“lua”))

Saída:

number

string

Nil

- Tipo de valor único, nil
- Variáveis, por padrão tem valor nil, até que sejam referenciadas
- Tem a única finalidade de apresentar uma variável que não é útil(não possui valor)
- Pode-se atribuir nil a uma variável para deletá-la

Boolean

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

Boolean

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

Number

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

```
x=0/0
print(x)
```

Nan é um valor numérico especial para representar valores indefinidos ou não representáveis

Saída:

```
-nan
```

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

Function

- Uma função pode ser passada como parâmetro de uma função, armazenada em uma variável ou retornada como resultado (Cidadão de 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 uma 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 Linguagem

- 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
- Lua é case sensitive

O uso dos comentários

- Comentários de linha são iniciados por, --
- Comentários de bloco são iniciados --[[e fechados com --]]

```
 9
10
11   a="Hello world" -- Um comentário
12   print(a)
13   --[[ Varios comentarios
14
15   muitos comentários
16
17   mais comentários --]]
18   print("Fim")
19
```

Palavras reservadas

- As palavras reservadas pela linguagem Lua são as seguintes:

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 como globais

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

Saída: 12.56

nil

Variáveis Locais

- Podem ser declaradas em qualquer lugar dentro de um bloco de comandos
- Necessário usa 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

- É possível criar um bloco com o comando do...end

```
1
2
3   x=4
4   print(x)
5   do
6   local x=1
7   local y=6
8   print(x)
9   print(y)
10  end
11  print(x)
12  print(y)
13
```

Saída:

4

1

6

4

nil

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

```
1  do
2      local x="Posso ser usado aqui!"
3      function funcaoInutil()
4          print(x)
5      end
6      funcaoInutil()
7      do
8          do
9          print(x)
10         end
11     end
12 end
13 print(x)
14
```

Saída:

Posso ser usado aqui!

Posso ser usado aqui!

nil

Atribuição

- Atribuição é feita pelo símbolo operador =
- Lua permite atribuição simples ou múltipla
- Pode-se utilizar ponto e vírgula (;) ou não no final de um comando

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

```
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
```

Atribuição

```
1
2
3   a,b=1,2
4   print(a)
5   print(b)
6   print('Trocando')
7   a,b=b,a
8   print(a)
9   print(b)
10
11
```

Saída:
1
2
Trocando
2
1

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 de dados e depois os valores

Tables, threads, e userdatas são comparadas 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

```
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

```
1  x=50
2  y="Tenho "
3  z=' fãs no orkut'
4  io.write(y..x..z)
5
```

Saída:

Tenho 50 fãs no orkut

Outros Operadores

- Operador de Comprimento
  - Denotado pelo operador unário #;

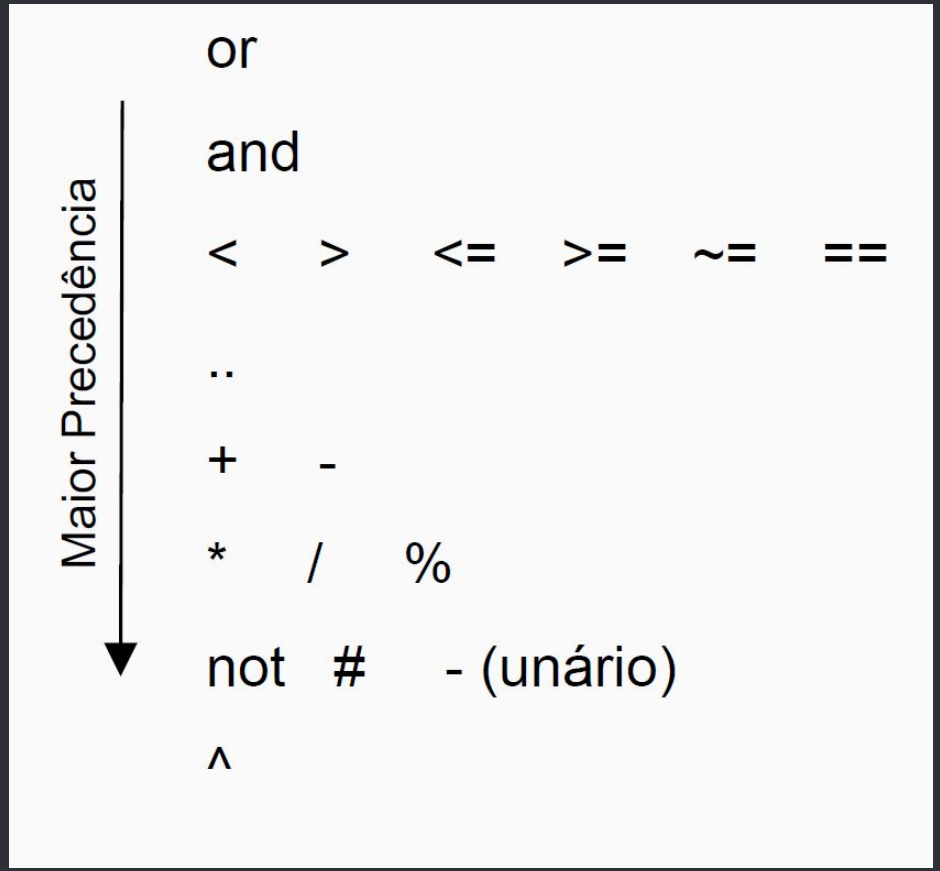
```
1  y="Tamanho"
2  print(y)
3  print(#y)
4
5
```

Saída:

Tamanho

7

Precedência



Estruturas de Controle

If Then Else

```
if (a == b) then
    print(“==”)
else
    print(“~=”)
end
```

```
if (a > b) then
    return a
elseif (b > a) then
    return b
else
    print(“==”)
```

While / Repeat Until

```
local s = 0
while (s < 10) do
    print(s)
    s = s + 1
end
```

```
local i = 100
repeat
print(i)
i = i - 1
until (i == 0)
```

For

```
for var = n_inicial, n_final, n_incremento do
    ...
end
```

Break / Return

- Break termina o loop em que está, dada uma condição, ou não, e o programa continua da onde parou após o loop;
- Return retorna um valor de uma função ou simplesmente termina ela. Toda função tem um return implícito, logo não é obrigatório escrever return no final da função.

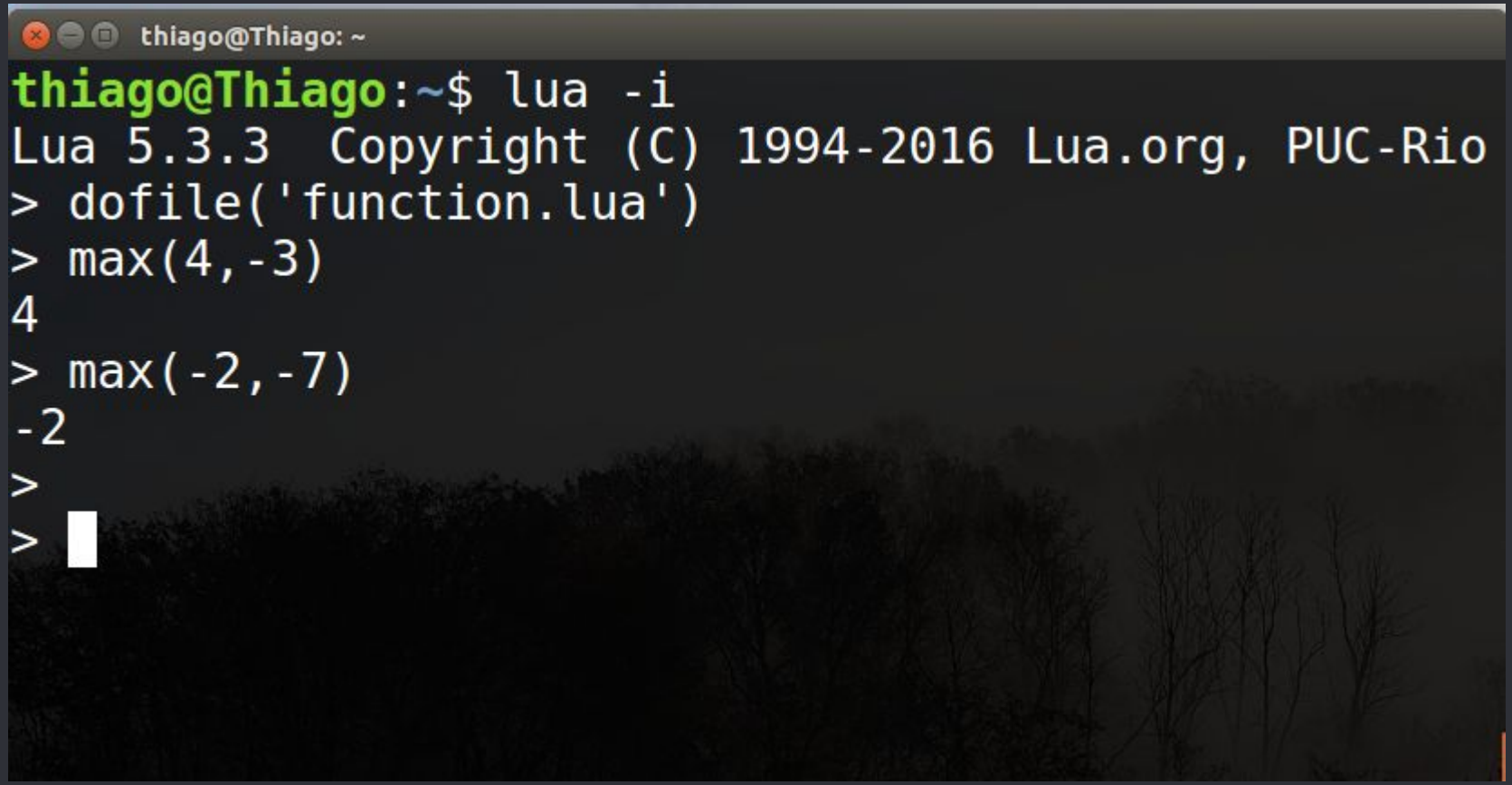
FUNÇÕES

Funções

- Funções são valores de primeira classe como todos os outros na linguagem;
- Logo elas podem ser guardadas em variáveis, passadas como argumentos e retornadas como resultado.

function.lua

```lua
function max(a, b)
    if (a > b) then
        return a
    else
        return b
    end
end

```

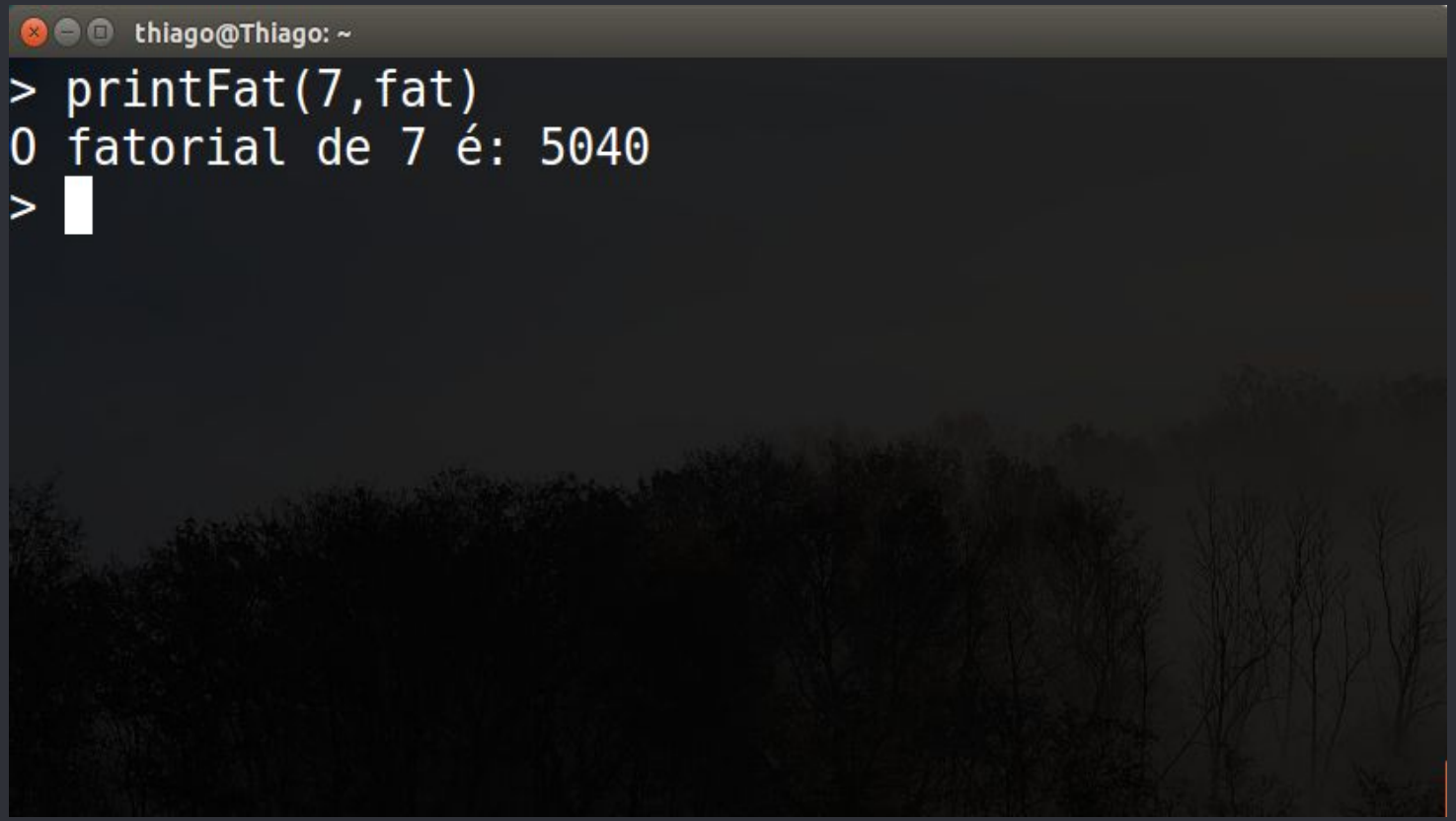

thiago@Thiago: ~
thiago@Thiago:~$ lua -i
Lua 5.3.3 Copyright (C) 1994-2016 Lua.org, PUC-Rio
> dofile('function.lua')
> max(4,-3)
4
> max(-2,-7)
-2
>
>

Funções

- Todas as funções em Lua são funções anônimas;
- O exemplo anterior mostrou um *syntactic sugar* para definir uma função e atribuí-la à uma variável global;
- Consequentemente uma função pode ser local.

function.lua

```lua
fat = function(n)
    if (n == 0) then
        return 1
    else
        return n*fat(n-1)
    end
end

function printFat(n,foo)
    io.write("O fatorial de ",n," é: ",foo(n),"\n")
end
```

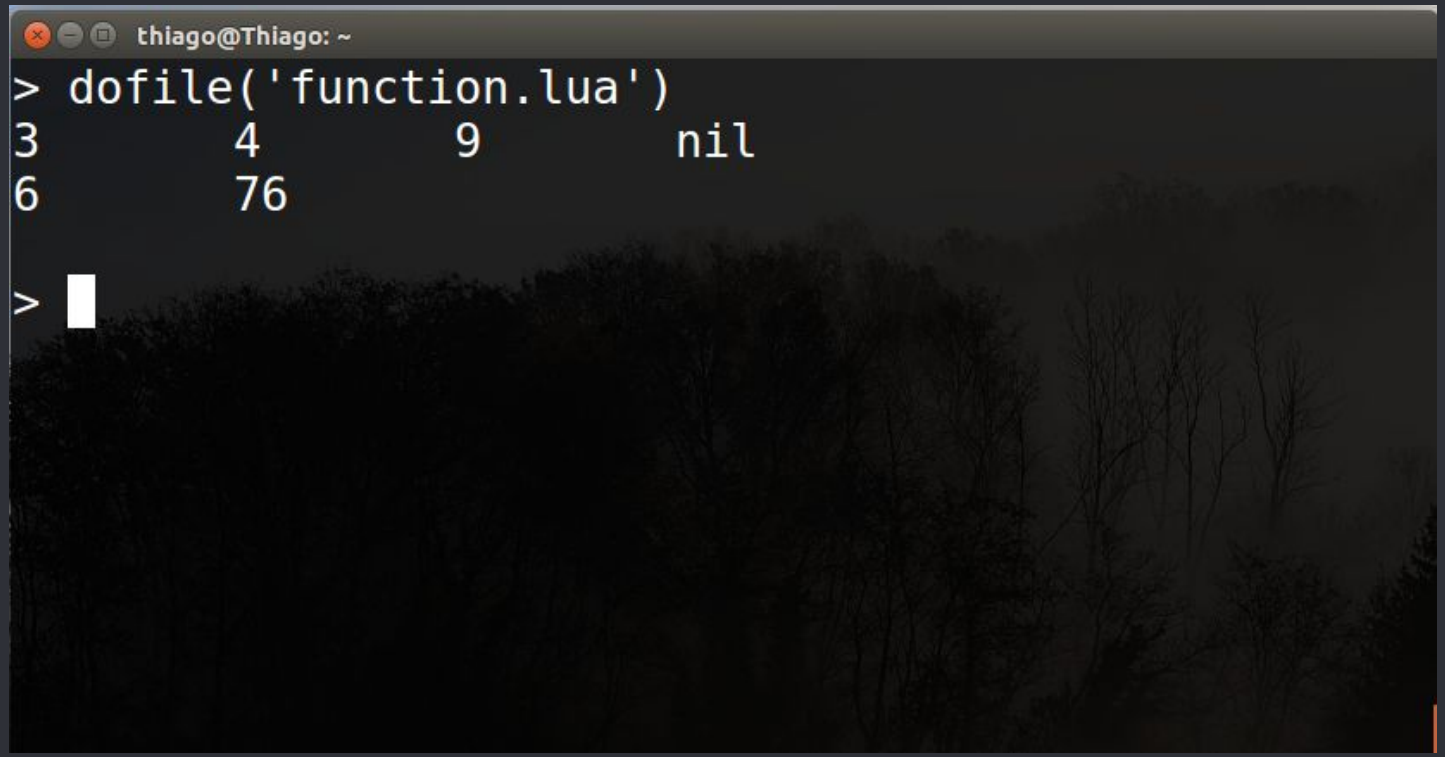

thiago@Thiago: ~
> printFat(7,fat)
0 fatorial de 7 é: 5040
>

Funções

- Funções também podem retornar múltiplos valores;
- Não há problema em passar mais argumentos do que os declarados para a função;
- A função pode receber um número variável de argumentos.

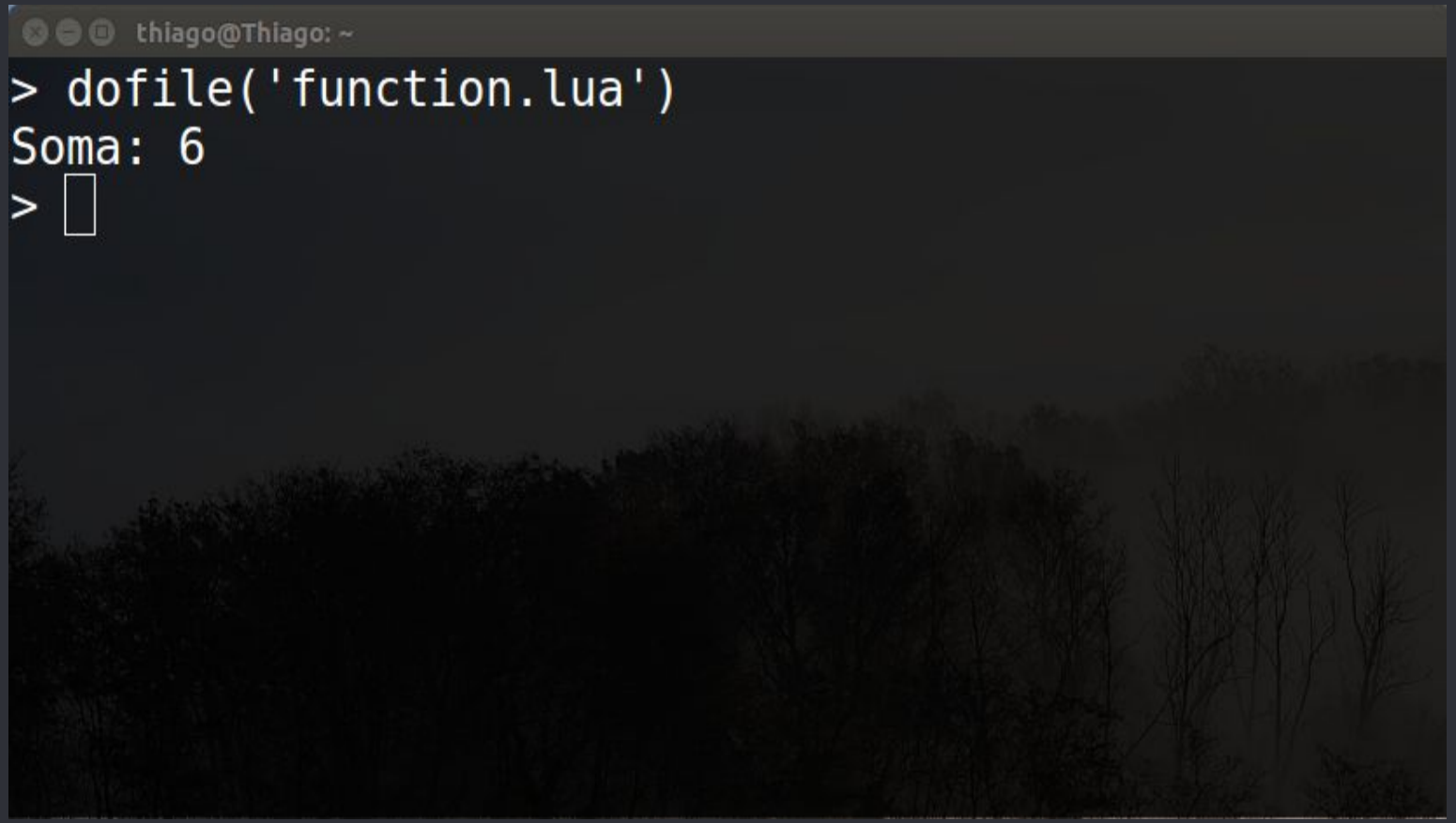
function.lua

```lua
function var2()
    return 3, 4, 9
end

function foo2(x,y)
    print(x,y)
end

a, b, c, d = var2()
print(a,b,c,d)
print(foo2(6,76,-9))
```


thiago@Thiago: ~
> dofile('function.lua')
3 4 9 nil
6 76
>

function.lua

```lua
function soma(...)
    local a = 0
    for i,v in ipairs{...} do
        a = a + v
    end
    return a
end
s = soma(2,3,-7,13,-5)
io.write("Soma: ",s,"\n")
```


thiago@Thiago: ~
> dofile('function.lua')
Soma: 6
>

Funções

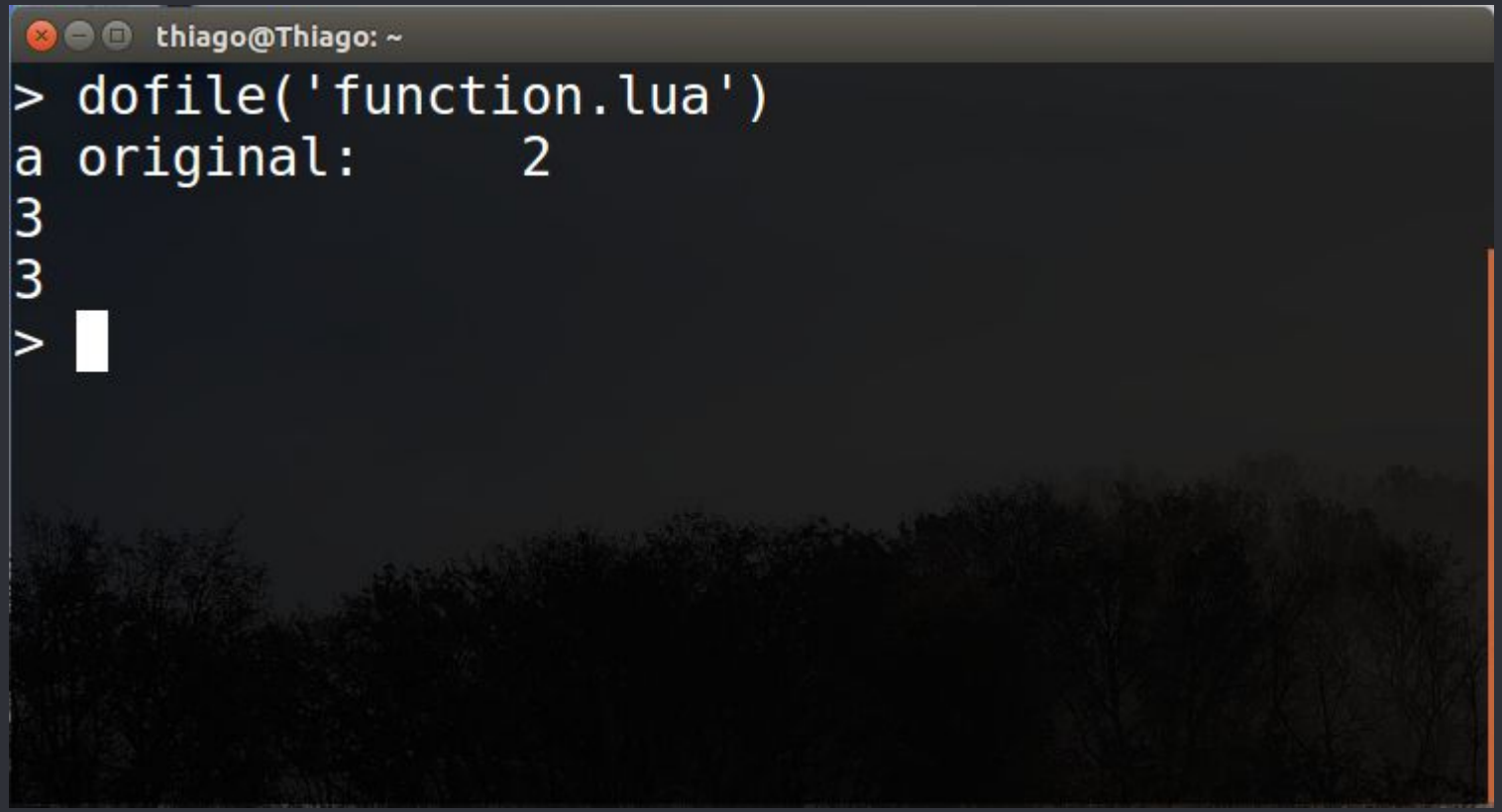
- A passagem de parâmetros é posicional;
- Tudo é passado por cópia, porém alguns tipos de valores são referência (function, table, userdata, thread);
- É fácil confundir com passagem por referência, porém isso será exemplificado em tables.

Funções

- Uma função pode acessar variáveis, mesmo locais, do escopo acima dela;
- Função pode ser aninhada dentro de outra função;
- Através disso definimos **closures**.

function.lua

```lua
local a = 2

function mod1(z)
    a = 3
end

function mod2(a)
    a = -17
end
print("a original:",a)
mod1(a); print(a)
mod2(a); print(a)
```

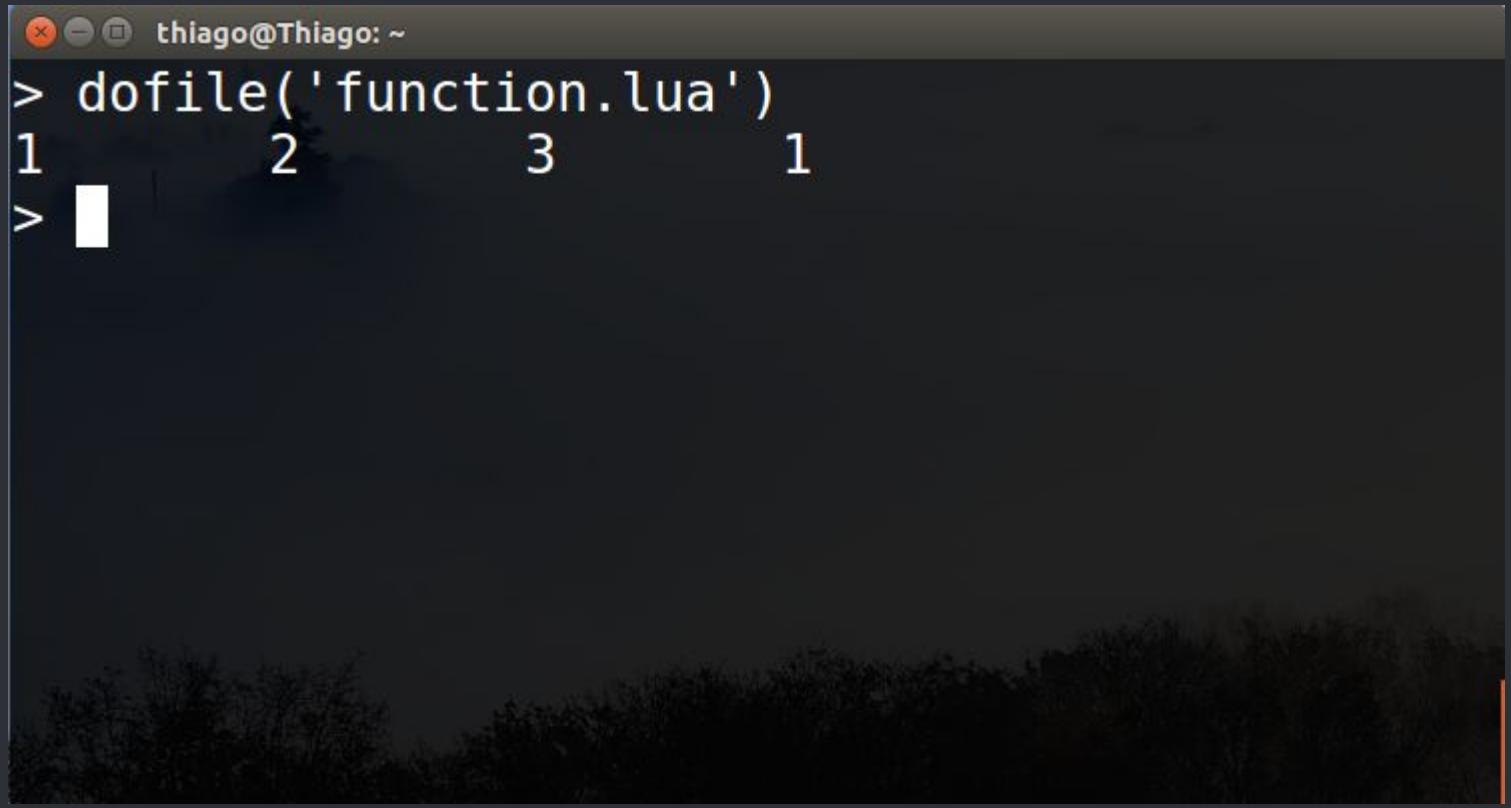

thiago@Thiago: ~
> dofile('function.lua')
a original: 2
3
3
>

Closures

- Conceito poderoso que poucas linguagens de programação suportam;
- Basicamente é uma função dentro de um bloco de código, ou de outra função, capaz de acessar variáveis de escopo acima, mas dentro do bloco.

function.lua

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

c=counter()
d=counter()
print(c(),c(),c(),d())
```


thiago@Thiago: ~
> dofile('function.lua')
1 2 3 1
>

Closures

- A função anônima dentro de **counter()** guarda o valor de **i**, mesmo o escopo de **i** tendo terminado;
- Similar a uma variável static em c, porém mais seguro;
- Cada nova instância para **counter()** cria uma nova cópia de **i;**

Closures

- **c()** e **d()** são closures diferentes sobre a mesma função.

TABELAS

Tabelas

- São a única estrutura de dados em Lua;
- Através das tabelas, se pode implementar as outras estruturas de dados conhecidas;
- O tipo table implementa uma array associativa, ou seja, seus índices podem ser strings, números, ou qualquer outro valor em Lua, exceto **nil**;

Tabelas

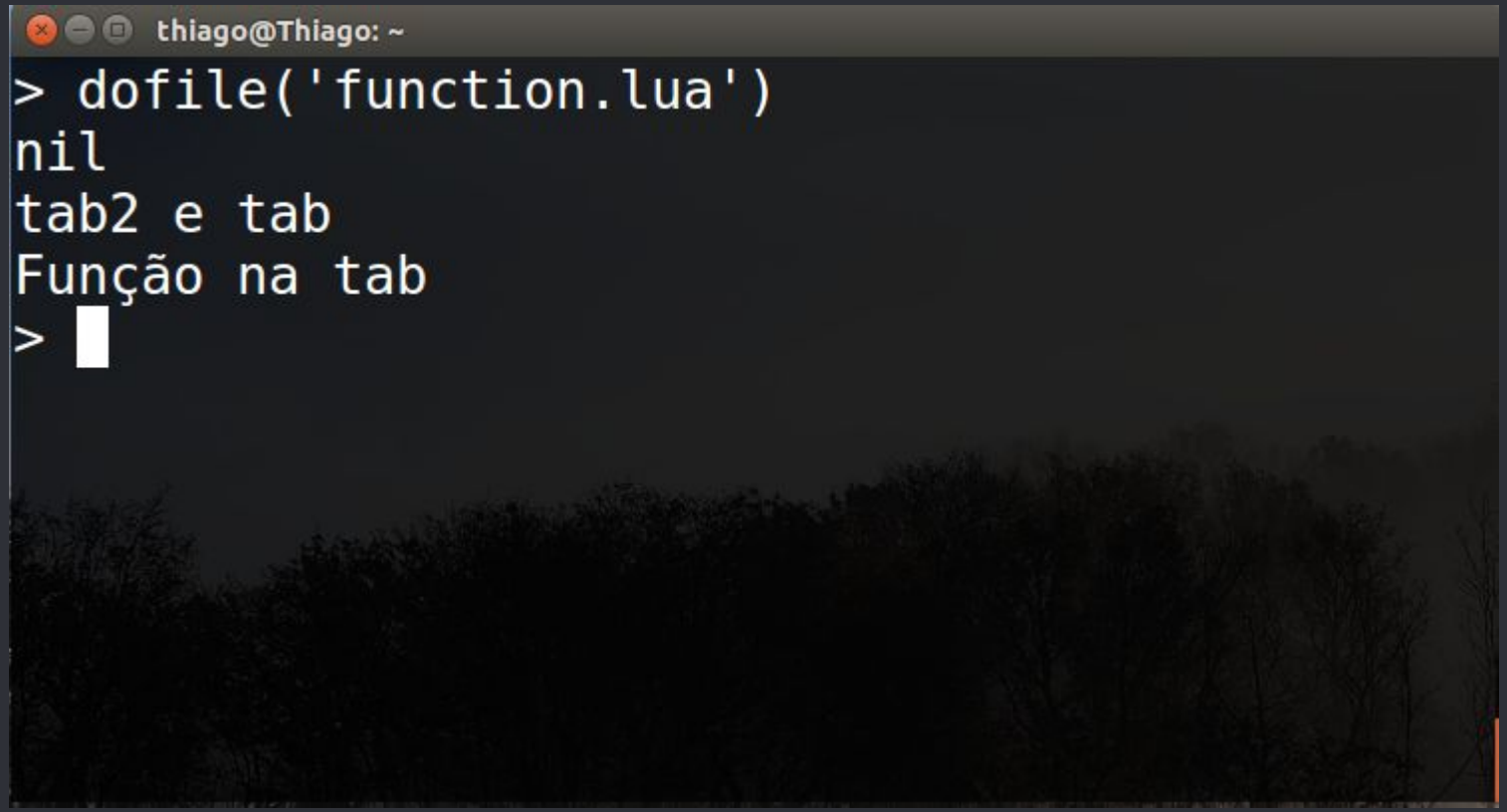
- Não apresentam tamanho fixo;
- Podem guardar desde funções, até outras tabelas e o valor nil;
- Tabelas são anônimas, como funções, não há relação fixa entre variáveis que guardam a tabela e a tabela em si;

Tabelas

- Quando um programa não apresenta mais referências para uma tabela, ela será deletada pelo coletor de lixo;
- Não existe declaração de tabelas em Lua;
- O primeiro índice numérico, se existir, começa em 1, ao invés de 0.

function.lua

```lua
tab = {}
tab[1] = "Bem vindo"
tab["foo"] = function()
        print("Função na tab") end

tab2 = tab
print(tab[2])
tab2[2] = "tab2 e tab"
print(tab[2])
tab.foo()
```

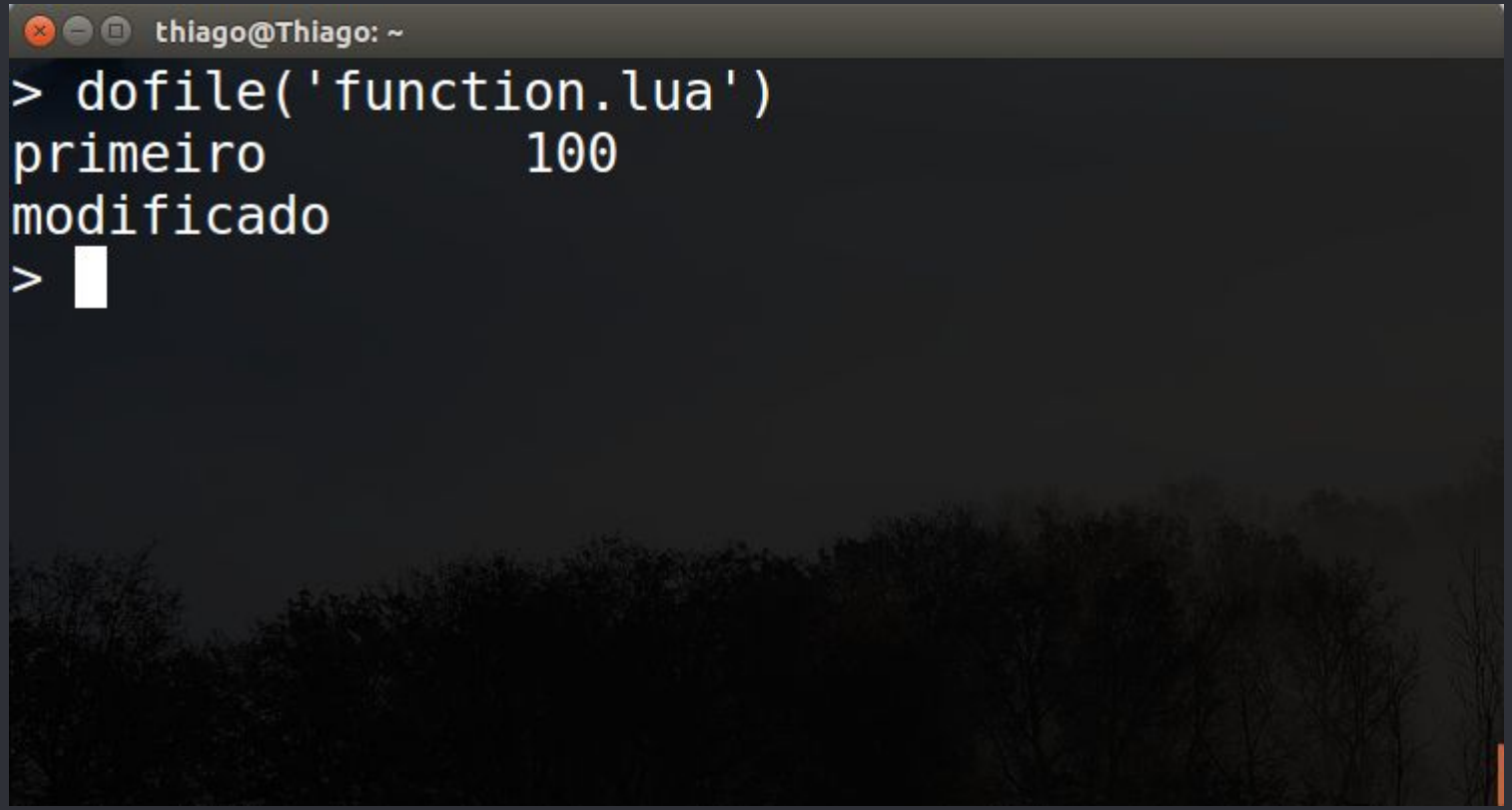

thiago@Thiago: ~
> dofile('function.lua')
nil
tab2 e tab
Função na tab
>

function.lua

```lua
dict = {
    "primeiro";
    foo=5;
    -17;
    dict2={100,101}
}

print(dict[1],dict.dict2[1])
```

function.lua

```lua
print(dict[1],dict.dict2[1])

function modt(table)
    table[1]="modificado"
    table={}
end

modt(dict)
print(dict[1])
```


thiago@Thiago: ~
> dofile('function.lua')
primeiro 100
modificado
>

Tabelas

A tabela **dict** não foi deletada em modt pois foi passado o valor da referência da tabela para a variável local **table** em **modt()**, logo após modificarmos realmente a tabela acessando seu índice, apenas deletamos sua referência em **table**.

Tabelas

- A biblioteca padrão Lua oferece duas funções para iterar sobre uma tabela:
  - **ipairs**, para iterar somente sobre índices numéricos: ipairs(t) itera sobre (1,t[1]),(2,t[2]) e assim por diante.
  - **pairs**, para iterar sobre todos e quaisquer índices.

ORIENTAÇÃO À OBJETOS

Orientação à Objetos

- Lua não oferece explicitamente mecanismos de OO;
- Porém eles podem ser implementados através das tabelas e funções;
- **Metatables** também são necessárias na implementação OO;

Orientação à Objetos

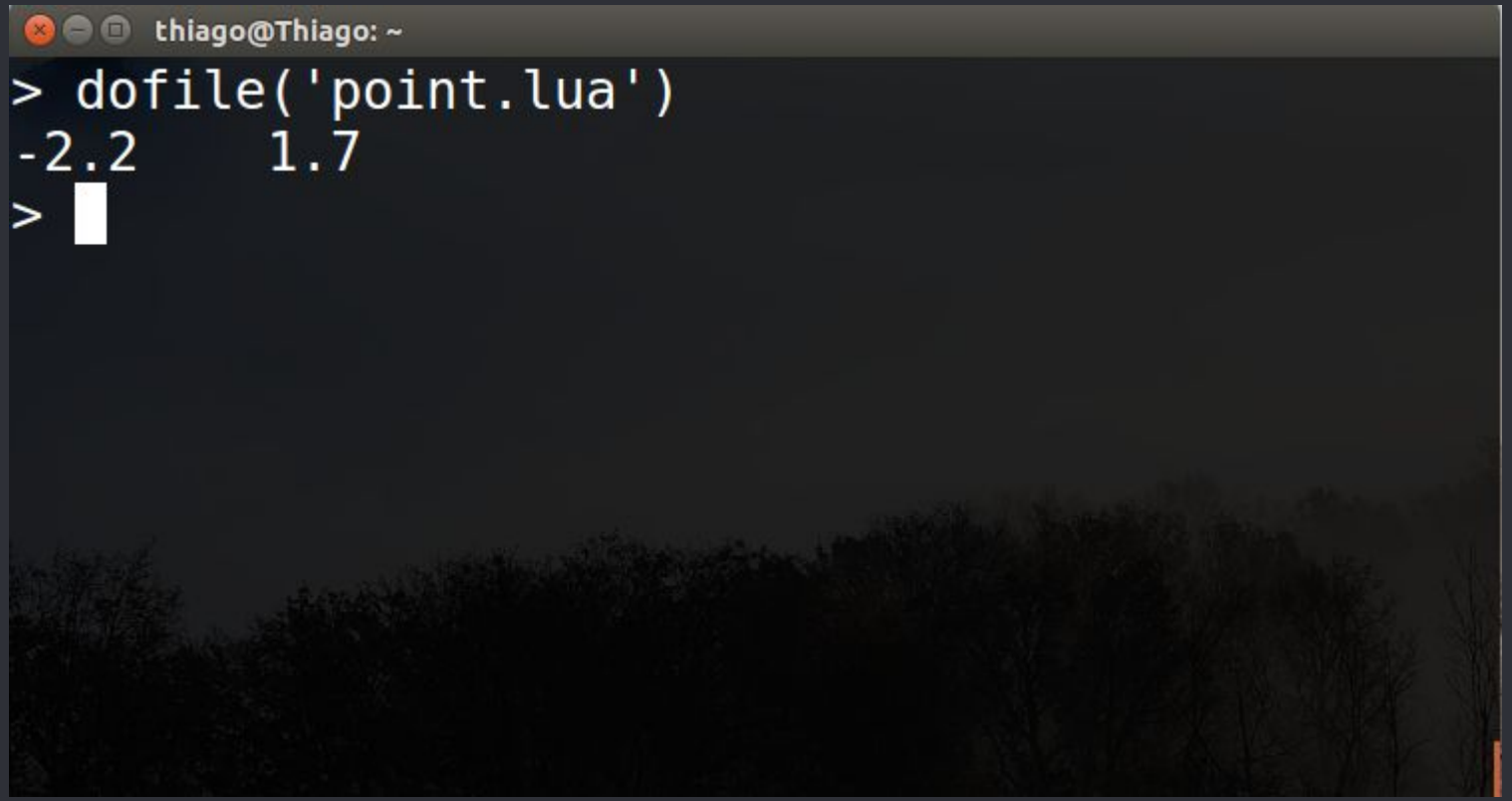
- Metatable é uma tabela que define o comportamento de um valor sobre alguma operação;
- Cada tabela e userdata tem suas metatables individuais, enquanto os outros tipos de valores em Lua compartilham uma única metatable para cada tipo;

Orientação à Objetos

- **Metamethods** são os campos da metatable que definem as operações para um tipo em específico;
- O parâmetro **self** é necessário nas implementações de métodos e construtores, similar ao **this**.

point.lua

```lua
function Point(self)
    self.x = tonumber(self.x) or 0.0
    self.y = tonumber(self.y) or 0.0
    self.z = tonumber(self.z) or 0.0
    return self
end

p = Point{x=2.0; y=4.5; z=-2.2}
q = Point{x=0.0; y=-1.5; z=1.7}

print(p.z,q.z)
```


thiago@Thiago: ~
> dofile('point.lua')
-2.2	1.7
>

Orientação à Objetos

- Para estender a semântica do nosso objeto do tipo **point** vamos adicionar a operação soma na **metatable** da nossa “classe”;
- Por padrão, os **metamethods** são nomeados com dois underscores e o nome da operação: __add

Orientação à Objetos

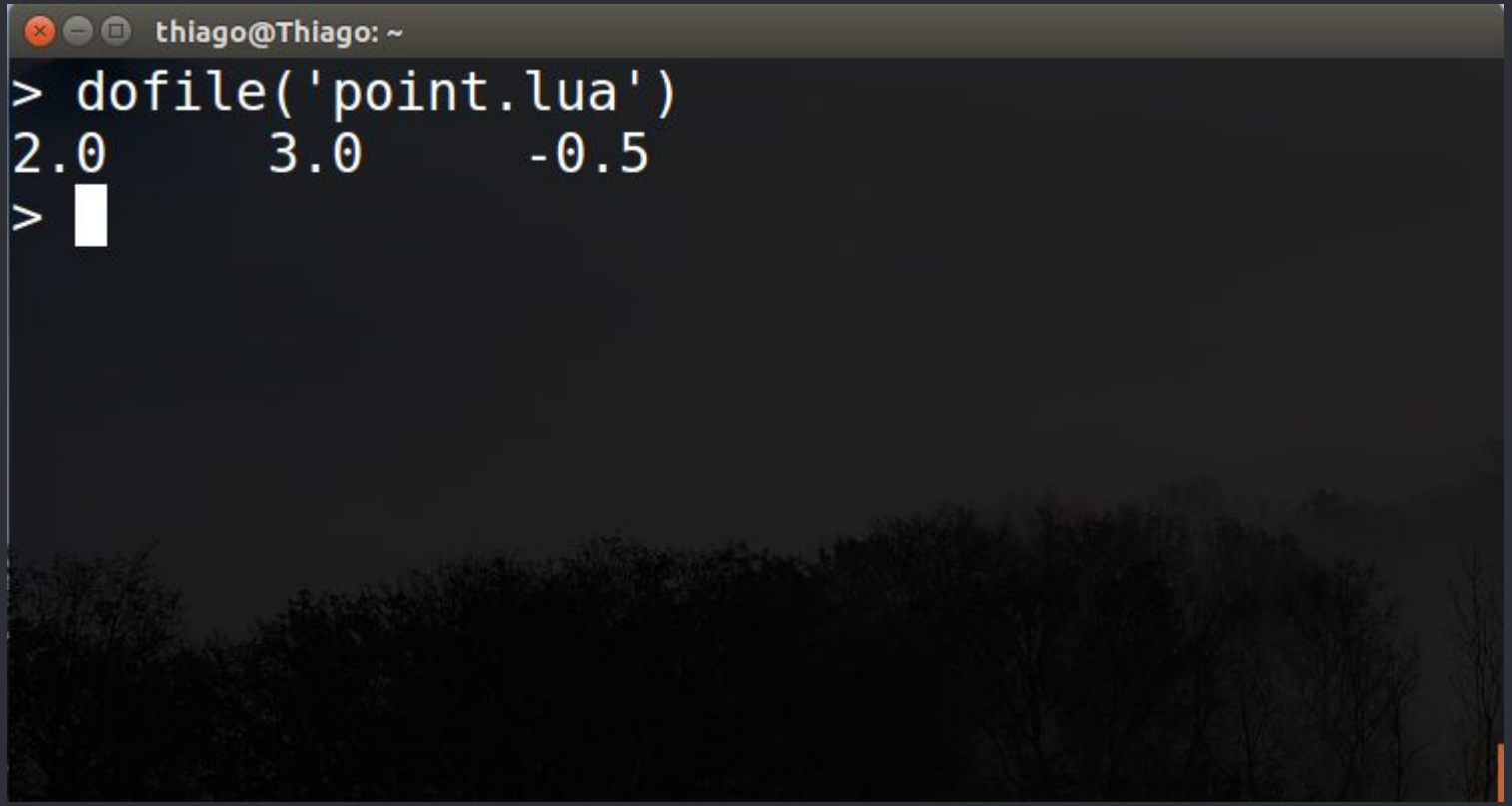
- Primeiro devemos criar um metatable com o nosso metamethod e então reescrever a função construtora do objeto.

point.lua

```lua
Point_metatable = {
    __add = function(p1,p2)
            return Point{x = p1.x+p2.x,
                         y = p1.y+p2.y,
                         z = p1.z+p2.z}
            end
}

function Point(self)
    self.x = tonumber(self.x) or 0.0
    self.y = tonumber(self.y) or 0.0
    self.z = tonumber(self.z) or 0.0
    setmetatable(self, Point_metatable)
    return self
end
```

point.lua

```
16
17  local p = Point{x=2.0; y=4.5; z=-2.2}
18  local q = Point{x=0.0; y=-1.5; z=1.7}
19  local r = p+q
20  print(r.x,r.y,r.z)
```


thiago@Thiago: ~
> dofile('point.lua')
2.0 3.0 -0.5
>

MODULARIZAÇÃO

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 muito tempo, não existia uma política bem definida para módulos.

Pela definição de módulo de Programming in Lua (2ª ed):

> *Um módulo é uma biblioteca que pode ser carregada usando-se require e que define um único nome global que armazena uma tabela*

Modularização

- 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:

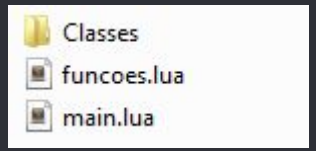
```
3 require 'math' --carrega modulo math
4 local sin = math.sin
5 local sqrt = math.sqrt
```

Modularização

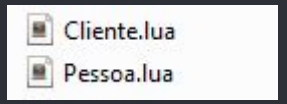
- Importando seu módulo de uma pasta do diretório atual:

```
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:** contém funções que manipulam cadeias de caracteres;
- **table:** manipulação de tabelas;

Modularização

- **math:** módulo com as funções matemáticas;
- **io:** biblioteca de entrada e saída (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 é simplesmente copiando o arquivo .lua, .so ou .dll para um diretório onde o interpretador de Lua poderá encontrá-lo.

Bibliotecas Padrão

- 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" (e.g. I/O);

Bibliotecas Padrão

- 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 Padrão

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);

Bibliotecas Padrão

- 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

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

```
1 function overload (arg1 , arg2 )
2     if type ( arg1 ) == 'string ' and type ( arg2 ) == 'string ' then
3         return arg1 .. arg2
4     elseif type ( arg1 ) == 'number ' and type ( arg2 ) == 'number ' then
5         return arg1 + arg2
6     end
7 end
8
9 a = overload ("10","10") --> 1010
10 b = overload (10 ,10) --> 20
```

Polimorfismo

Inclusão

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

CONCORRÊNCIA

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;

Concorrência

- É semelhante a uma thread, há uma linha de execução compartilhando sua pilha de execução com outras co-rotinas.

- A diferença é que, diferentes threads executam simultaneamente, enquanto apenas uma co-rotina executa por vez.

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).

- As funções que manipulam co-rotinas estão agrupadas na tabela coroutine.

Concorrência

- 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”.

Concorrência

- A função *coroutine.status* retorna o estado da co-rotina

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

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

Concorrência

- 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.

EXCEÇÕES

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;

Exceções

```
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 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.

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

Referências

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