

# Linguagens de Programação

Prof. Miguel Elias Mitre Campista

<http://www.gta.ufrj.br/~miguel>

## Parte II

Programação em Linguagens Estruturadas

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Quais Linguagens?

- Lua
- Perl

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Linguagem Lua

- Criada em 1993 na PUC-Rio
- Linguagem de script dinâmica
  - Semelhante a Python, PHP e Ruby
- Possui simplicidade de codificação, eficiência e portabilidade
- Possui possibilidade de embutir o interpretador em uma aplicação C
- Tamanho pequeno
  - Núcleo da linguagem mais bibliotecas ocupa menos de 200k
    - Importante para arquiteturas com recursos limitados

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Linguagem Lua

- É uma linguagem dinâmica...
  - Interpretação dinâmica
    - Linguagem capaz de executar trechos de código criados dinamicamente no mesmo ambiente de execução
      - Ex.: função `loadstring`

```
f = loadstring ("i = i + 1")
i = 0
f (); print (i) -- Imprime 1
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Linguagem Lua

- É uma linguagem dinâmica...
  - Tipagem dinâmica forte
    - Tipagem **dinâmica** faz verificação de tipos em tempo de execução e não em tempo de compilação
      - Além disso, não faz declaração de tipos no código
    - Tipagem **forte** não aplica uma operação a um tipo incorreto
  - Gerência automática de memória dinâmica
    - Memória não precisa ser tratada explicitamente no programa
      - Ex.: Alocação e liberação de memória

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Linguagem Lua

- Possui propósito geral
  - Pode ser utilizada em...
    - Pequenos scripts e sistemas complexos
- Principais aplicações
  - Desenvolvimento de jogos
    - Ex.: "World of Warcraft" e "The Sims"
  - Middleware do Sistema Brasileiro de TV Digital
    - Ex.: Projeto "Ginga"
  - Software comercial
    - Ex.: "Adobe Photoshop Lightroom"
  - Software para Web
    - Ex.: "Publique!"

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Linguagem Lua

- Trecho
  - Peça de código em Lua
- Compila códigos para máquina virtual (MV)
- Depois de compilado, Lua executa o código com o interpretador para a MV
  - Interpretador: lua
    - Compila e executa o código
      - lua <arq-codigo>
  - Compilador: luac
    - Apenas compila
      - luac -o <nome-arq-compilado> <arq-codigo>

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Primeiro Exemplo em Lua

- Programa: HelloWorld.lua

```
print 'Hello, world!'
```

- Compilação+Execução: lua HelloWorld.lua

```
shell>$ lua HelloWorld.lua
Hello, world!
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Primeiro Exemplo em Lua

- Programa: HelloWorld.lua

```
print 'Hello, world!'
```

- Compilação seguida de execução: luac -o l HelloWorld.lua

```
shell>$ luac -o l HelloWorld.lua
shell>$ lua l
Hello, world!
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Primeiro Exemplo em Lua

- Programa: HelloWorld.lua

```
print 'Hello, world!'
```

- Compilação seguida de execução: luac -o l HelloWorld.lua

```
shell>$ luac -o l HelloWorld.lua
shell>$ lua l
Hello, world!
shell>$
```

Distribuição de Lua para Windows: "Lua for windows"  
<http://code.google.com/p/luaforwindows/>

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Primeiro Exemplo em Lua

- Modo pela linha de comando:

```
shell>$ lua -e "print 'Hello, world!'"
Hello, world!
shell>$
```

- Modo interativo:

```
shell>$ lua
> print "Hello, world!"
Hello, world!
>
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
  - Por padrão, as variáveis são sempre globais (Escopo léxico)
    - Para indicar variáveis locais, usa-se a palavra-chave: `local`
- Tipos
  - Determinados dinamicamente, dependendo do valor que está sendo armazenado
    - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
      - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
  - Por padrão, as variáveis são sempre globais (Escopo léxico)
    - Para indicar variáveis locais, usa-se a palavra-chave: `local`
- Tipos
  - Determinados dinamicamente, dependendo do valor que está sendo armazenado
    - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
      - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`

Semelhante ao NULL, significada ausência de valor

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
  - Por padrão, as variáveis são sempre globais (Escopo léxico)
    - Para indicar variáveis locais, usa-se a palavra-chave: `local`
- Tipos
  - Determinados dinamicamente, dependendo do valor que está sendo armazenado
    - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
      - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`

Variável booleana

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
  - Por padrão, as variáveis são sempre globais (Escopo léxico)
    - Para indicar variáveis locais, usa-se a palavra-chave: `local`
- Tipos
  - Determinados dinamicamente, dependendo do valor que está sendo armazenado
    - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
      - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`

Ponto flutuante (pode ser usada para representar um inteiro)

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
  - Por padrão, as variáveis são sempre globais (Escopo léxico)
    - Para indicar variáveis locais, usa-se a palavra-chave: `local`
- Tipos
  - Determinados dinamicamente, dependendo do valor que está sendo armazenado
    - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
      - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`

Cadeia de caracteres: `'cadeia'`, `"cadeia"` ou `[[cadeia]]`

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
  - Por padrão, as variáveis são sempre globais (Escopo léxico)
    - Para indicar variáveis locais, usa-se a palavra-chave: `local`
- Tipos
  - Determinados dinamicamente, dependendo do valor que está sendo armazenado
    - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
      - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`

Representa funções

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
    - Por padrão, as variáveis são sempre globais (Escopo léxico)
      - Para indicar variáveis locais, usa-se a palavra-chave: `local`
  - Tipos
    - Determinados dinamicamente, dependendo do valor que está sendo armazenado
      - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
        - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`
- Tipo para tabelas (arrays, conjuntos, grafos etc.)

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
    - Por padrão, as variáveis são sempre globais (Escopo léxico)
      - Para indicar variáveis locais, usa-se a palavra-chave: `local`
  - Tipos
    - Determinados dinamicamente, dependendo do valor que está sendo armazenado
      - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
        - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`
- Área de memória sem operação pré-determinada

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
    - Por padrão, as variáveis são sempre globais (Escopo léxico)
      - Para indicar variáveis locais, usa-se a palavra-chave: `local`
  - Tipos
    - Determinados dinamicamente, dependendo do valor que está sendo armazenado
      - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
        - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`
- Área de memória sem operação pré-determinada

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Variáveis em Lua

- Escopo
  - Por padrão, as variáveis são sempre globais (Escopo léxico)
    - Para indicar variáveis locais, usa-se a palavra-chave: `local`
- Tipos
  - Determinados dinamicamente, dependendo do valor que está sendo armazenado
    - Variáveis podem armazenar qualquer um dos tipos básicos de Lua
      - `nil`, `boolean`, `number`, `string`, `function`, `table` e `userdata`
        - » `function`, `table` e `userdata` armazenam uma referência

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Segundo Exemplo em Lua

- Programa: `tipos.lua`

```
local a = 3
print (type (a)) -- imprime "number"
a = "lua"
print (type (a)) -- imprime "string"
a = true
print (type (a)) -- imprime "boolean"
a = print -- "a" agora é a função "print"
a (type (a)) -- imprime "function"
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Segundo Exemplo em Lua

- Programa: `tipos.lua`

```
local a = 3
print (type (a)) -- imprime "number"
a = "lua"
print (type (a)) -- imprime "string"
a = true
print (type (a)) -- imprime "boolean"
a = print -- "a" agora é a função "print"
a (type (a)) -- imprime "function"
```

Variável "a" é declarada local

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Segundo Exemplo em Lua

- Programa: tipos.lua

```
local a = 3
print (type (a)) -- imprime "number"
a = "lua"
print (type (a)) -- imprime "string"
a = true
print (type (a)) -- imprime "boolean"
a = print -- "a" agora é a função "print"
a (type (a)) -- imprime "function"
```

Função type que retorna o tipo da variável

## Segundo Exemplo em Lua

- Programa: tipos.lua

```
local a = 3
print (type (a)) -- imprime "number"
a = "lua"
print (type (a)) -- imprime "string"
a = true
print (type (a)) -- imprime "boolean"
a = print -- "a" agora é a função "print"
a (type (a)) -- imprime "function"
```

Comentário

## Segundo Exemplo em Lua

- Programa: tipos.lua

```
local a = 3
print (type (a)) -- imprime "number"
a = "lua"
print (type (a)) -- imprime "string"
a = true
print (type (a)) -- imprime "boolean"
a = print -- "a" agora é a função "print"
a (type (a)) -- imprime "function"
```

- Interpretação: lua tipos.lua

```
shell>$ lua tipos.lua
number
string
boolean
function
shell>$
```

## Inicialização de Variáveis

- Programa: initvar.lua

```
x = 1 -- x recebe 1
b, c = "bola", 3 -- b recebe o valor "bola" e c o valor 3
print (b, y)

a, b, sobrei = 1, 2 -- número de variáveis é maior
print (a, b, sobrei)

x, y = "bola", "casa", "sobrei" -- número de valores é maior
print (x, y)

x, y = y, x -- swap
print (x, y)
```

```
shell>$ lua initvar.lua
bola nil
1 2 nil
bola casa
casa bola
shell>$
```

## Escopo de Variáveis

- Programa: escopo.lua

```
local x = 5
local y
print (x, y)

do -- Início de um bloco
  local x = 10
  y = 1
  print (x, y)
end -- Término do bloco

print (x, y)
```

```
shell>$ lua escopo.lua
5 nil
10 1
5 1
shell>$
```

## Escopo de Variáveis

- Programa: escopo.lua

```
local x = 5
local y
print (x, y)

do -- Início de um bloco
  local x = 10
  y = 1
  print (x, y)
end -- Término do bloco

print (x, y)
```

```
shell>$ lua escopo.lua
5 nil
10 1
5 1
shell>$
```

Em Lua, mesmo variáveis com escopo global podem ser declaradas locais. O acesso a variáveis locais é mais eficiente que o acesso a variáveis globais

## Escopo de Variáveis

- Programa: escopo.lua

```
local x = 5
local y
print (x, y)
```

```
do -- Início de um bloco
  local x = 10
  y = 1
  print (x, y)
end -- Término do bloco
```

```
print (x, y)
```

```
shell>$ lua escopo.lua
5      nil
10     1
5      1
shell>$
```

do-end delimitam um bloco, mas qualquer outra estrutura de controle (if, while, for) também poderia ser usada

## Operadores em Lua

- Relacionais

- <, >, <=, >=, ==, ~=

- Operadores retornam true ou false (0 é tipo number)
- Negação da igualdade: "~="

- Lógicos

- and, or, not

## Operadores em Lua

- Relacionais

- <, >, <=, >=, ==, ~=

- Operadores retornam true ou false
- Negação da igualdade: "~="

- Lógicos

- and, or, not

Arquivo: var.lua

```
print (34 or nil) --> 34
print (not 34) --> false
print (true and 0) --> 0
print (not not 0) --> true
print (false or "lua") --> lua
print (n and "33" or "34") --> 34
x = v or 100
print (x) --> █
```

and: retorna o primeiro se for false ou nil ou o segundo, caso contrário

or: retorna o primeiro operando que não for nil ou false

not: retorna sempre um valor booleano

## Operadores em Lua

- Relacionais

- <, >, <=, >=, ==, ~=

- Operadores retornam true ou false
- Negação da igualdade: "~="

- Lógicos

- and, or, not

Arquivo: var.lua

```
print (34 or nil) --> 34
print (not 34) --> false
print (true and 0) --> 0
print (not not 0) --> true
print (false or "lua") --> lua
print (n and "33" or "34") --> 34
x = v or 100
print (x) --> █
```

```
shell>$ lua var.lua
34
false
0
true
lua
34
100
shell>$
```

## Operadores em Lua

- Pra que eu poderia usar a função abaixo em Lua?

```
function initx (v)
  x = v or 100
end
```

## Operadores em Lua

- Pra que eu poderia usar a função abaixo em Lua?

```
function initx (v)
  x = v or 100
end
```

Função para inicializar x com valor padrão (100) caso v não seja atribuído.

## Operadores em Lua

Arquivo: defaultInit.lua

```
function initx (v)
    x = v or 100
end

initx ()
print (x)

initx (2)
print (x)
```

```
miguel@pegasus-linux:~$ lua defaultInit.lua
100
2
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Tabelas em Lua

- Uso do operador {}
  - Tabela vazia: t = {}
  - Tabela com três elementos: t = {4, "lua", false}
  - Tabela associativa (chave e valor): t = {x=4, y="l", z=false}

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Tabelas em Lua

- Uso do operador {}
  - Tabela vazia: t = {}
  - Tabela com três elementos: t = {4, "lua", false}
  - Tabela associativa (chave e valor): t = {x=4, y="l", z=false}

Arquivo: tabela.lua

```
-- Tabela
t = {4, "lua", false}
print (t[1], t[2], t[3])
print (type(t[1]), type(t[2]), type(t[3]))

-- Tabela Associativa
t = {x=4, y="lua", z=false}
print (t[1], t[2], t[3])
print (t["x"], t["y"], t["z"])
print (t.x, t.y, t.z)
```

## Tabelas em Lua

- Uso do operador {}
  - Tabela vazia: t = {}
  - Tabela com três elementos: t = {4, "lua", false}
  - Tabela associativa (chave e valor): t = {x=4, y="l", z=false}

Arquivo: tabela.lua

```
-- Tabela
t = {4, "lua", false}
print (t[1], t[2], t[3])
print (type(t[1]), type(t[2]), type(t[3]))

-- Tabela Associativa
t = {x=4, y="lua", z=false}
print (t[1], t[2], t[3])
print (t["x"], t["y"], t["z"])
print (t.x, t.y, t.z)
```

```
shell->$ lua tabela.lua
4      lua      false
number string  boolean
nil    nil        nil
4      lua      false
4      lua      false
shell->$
```

## Tabelas em Lua

- Programa: tabela2.lua (Variáveis do tipo table armazenam referências)

```
local tab1 = {} -- cria uma tabela
local tab2 = {}

tab1.x = 33 -- associa valor 33 com chave "x"
tab2.x = 33

print (tab1 == tab2) -- imprime "false"

tab1 = tab2
print (tab1 == tab2) -- imprime "true"

tab2.x = 20
print (tab1.x)
-- imprime 20, pois tab1 e tab2 se referem ao mesmo valor
```

```
shell->$ lua tabela2.lua
false
true
20
shell->$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Tabelas em Lua

- Inserção em tabelas

Arquivo: insereTable.lua

```
local t = {x=100, y=200, w=50}
print (t ["y"], t.w)

t [100] = true
t ["a"] = "A"
t.cursor = "Lua"
print (t ["y"], t.w, t[100], t.a, t ["curso*"])
```

```
miguel@pegasus-linux:~$ lua insereTable.lua
200 50
200 50 true A Lua
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Tabelas em Lua

- Inserção em tabelas

Arquivo: `insereTable.lua`

```
local t = {x=100, y=200, w=50}
print (t ["*y"], t.w)

t [100] = true
t ["*a"] = "A"
t.curso = "Lua"
print (t ["*y"], t.w, t[100], t.a, t ["curso*"])
```

```
miguel@pegasus-linux:~$ lua insereTable.lua
200 50
200 50 true A Lua
```

A inserção em uma tabela pode ser feita a partir de uma atribuição

Linguagem

## Tabelas em Lua

- Uso de função em tabelas

Arquivo: `funcaoTable.lua`

```
function hello (nome)
    print ("Hello,", nome)
end

--> Declaração e inicializando da tabela t
local t = {1, 2, hello}
print (t [1], t [2], t [3])
t [3] ("Miguel")

--> Reinizalizando da tabela t
t = {x = 1, y = hello}
print (t.x, t ["*x"], t.y, t ["*y"])
t.y ("Elias")
t ["*y"] ("Campista")
```

```
shell->$ lua funcaoTable.lua
1 2 function: 0x1cb2bb0
Hello, Miguel
1 1 function: 0x1cb2bb0
function: 0x1cb2bb0
Hello, Elias
Hello, Campista
Hello, Miguel
shell->$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Tabelas em Lua

- Uso de função em tabelas

Arquivo: `funcaoTable.lua`

```
function hello (nome)
    print ("Hello,", nome)
end

--> Declaração e inicializando da tabela t
local t = {1, 2, hello}
print (t [1], t [2], t [3])
t [3] ("Miguel")

--> Reinizalizando da tabela t
t = {x = 1, y = hello}
print (t.x, t ["*x"], t.y, t ["*y"])
t.y ("Elias")
t ["*y"] ("Campista")
```

```
shell->$ lua funcaoTable.lua
1 2 function: 0x1cb2bb0
Hello, Miguel
1 1 function: 0x1cb2bb0
function: 0x1cb2bb0
Hello, Elias
Hello, Campista
Hello, Miguel
shell->$
```

A inserção de uma função em uma tabela pode ser feita na própria inicialização

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Tabelas em Lua

- Inserção em tabelas associativas

Arquivo: `assocTable.lua`

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

--> Combinação de chaves
t = {[1] = 3, [2] = 4, [3] = soma}
print ("Resultado = ", t [3] (t [1], t[2]))
```

```
shell->$ lua assocTable.lua
Resultado = 7
shell->$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Tabelas em Lua

- Inserção em tabelas associativas

Arquivo: `assocTable.lua`

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

--> Combinação de chaves
t = {[1] = 3, [2] = 4, [3] = soma}
print ("Resultado = ", t [3] (t [1], t[2]))
```

Tabela associativa com chave inteira

```
shell->$ lua assocTable.lua
Resultado = 7
shell->$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Estruturas de Controle em Lua

- Condição

```
if x > 10 then
    print ("x > 10")
elseif x > 5 then
    print ("5 < x < 10")
else
    print ("x < 5")
end
```

- Laço

```
while x < 10 do
    x = x + 1
end
```

```
-- valor inicial, cond. de contorno e passo
for x=1, 10, 1 do
    print (x)
end

for x=1, 10 do --Passo igual a 1 pode omitir
    print (x)
end

for x=10, 1, -1 do
    print (x)
end
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista



## Estruturas de Controle em Lua

- `for` genérico: Percorre valores com uma fç iteradora
  - `ipairs`: percorre índices de um array
  - `pairs`: percorre chaves de uma tabela
  - `io.lines`: percorre linhas de um arquivo

```
a1 = {1, 3, 5}
for i, v in ipairs (a1) do
  print (v)
end

a2 = {x=1, y=3, z=5}
for k, v in pairs (a2) do
  print (k, v)
end

-- Nome do arquivo: arquivo.txt
for l in io.lines ("arquivo.txt") do
  print (l)
end
```

### Programa iteradores.lua

```
shell->$ lua iteradores.lua
1
3
5
x      1
y      3
z      5
aluno
eletrônica
linguagens
shell->$
```

## Estruturas de Controle em Lua

- Qual `for` genérico pode ser usado para imprimir todos os elementos (`ipairs`, `pairs`, `io.lines`)?

```
local t = {x=100, y=200, w=50}
print (t ["y"], t.w)

t [100] = true
t ["a"] = "A"
t.curso = "Lua"
print (t ["y"], t.w, t[100], t.a, t ["curso"])
```

## Estruturas de Controle em Lua

- Qual `for` genérico pode ser usado para imprimir todos os elementos (`ipairs`, `pairs`, `io.lines`)?

```
local t = {x=100, y=200, w=50}
print (t ["y"], t.w)

t [100] = true
t ["a"] = "A"
t.curso = "Lua"
print (t ["y"], t.w, t[100], t.a, t ["curso"])

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

```
miguel@pegasus-linux:~$ lua insereTable.lua
200 50
200 50 true A Lua
y 200
100 true
curso Lua
a A
w 50
x 100
```

## Funções em Lua

- Declaração de funções

– function

```
function nome-da-funcao (arg1, arg2, ..., argn)
  Corpo da função
end
```

• Ex.: function fatorial (n)

- Funções podem receber e retornar n parâmetros

```
function nome-da-funcao (arg1, arg2, ..., argn)
  corpo da função
  return par1, par2, ..., parn
end
x1, x2, ..., xn = nome-da-funcao(arg1, arg2, ..., argn)
```

## Funções em Lua

```
function impar (n)
  if n == 0 then
    return false
  else
    return par (n - 1)
  end
end

function par (n)
  if n == 0 then
    return true
  else
    return impar (n - 1)
  end
end

local n = io.read ("*number")
print (par (n))
```

```
shell->$ lua parimpar.lua
2
true
shell->$
```

## Funções em Lua

- Funções podem ainda receber um número variável de parâmetros

– Uso das reticências

```
function nome-da-funcao (...)
  for i, v in ipairs {...} do
    print (v)
  end
end
```

## Funções em Lua

```
function printArray (...)  
  for i, v in ipairs {...} do  
    print (v)  
  end  
end  
printArray (1, 2, 3)  
printArray ("a", "b", "c", "d", "e")
```

```
shell->$ lua funcoes.lua  
1  
2  
3  
a  
b  
c  
d  
e  
shell->$
```

## Funções em Lua

- Relembrando a função loadstring...

```
-- Lê uma string do teclado  
local n = io.read ("*line")  
print (n)  
  
f = loadstring (n)  
i = 0  
f (); print (i)
```

```
shell->$ lua exemploLoadstring.lua  
i= i + 2  
i= i + 2  
2  
shell->$
```

## Retorno das Funções

- Nem sempre todos os valores retornados são usados
  - Em uma lista de funções, apenas o primeiro valor retornado de cada membro da lista é usado

```
function func (a, b)  
  local x = a or 0  
  local y = b or 1  
  return x + y, x * y  
end
```

```
shell->$ lua retornoFuncao.lua  
3 7 11 30  
shell->$
```

```
a, b, c, d = func (1, 2), func (3, 4), func (5, 6)  
print (a, b, c, d)
```

## Uso de Funções como Argumento

- Funções podem ser passadas como argumentos para outras funções
  - Pode-se também retornar funções

```
function map (f, t)  
  for k, v in pairs (t) do  
    t [k] = f (v)  
  end  
  return t  
end
```

```
shell->$ lua argFuncao.lua  
2  
3  
4  
shell->$
```

```
function inc (v)  
  return v + 1  
end  
  
local vec = {1, 2, 3}  
  
map (inc, vec)  
  
for k, v in pairs (vec) do  
  print (vec [k])  
end
```

## Passagem de Parâmetro para Função

- O que vai ser impresso na tela?

```
function soma (a)  
  a = a + 2  
end
```

(a) local a = 2  
print (a)  
soma (a)  
print (a)

```
function soma (a)  
  a[1] = a[1] + 2  
end
```

(b) local a = {2}  
print (a[1])  
soma (a)  
print (a[1])

```
shell->$ lua soma.lua  
2  
2  
shell->$
```

```
shell->$ lua soma.lua  
2  
4  
shell->$
```

## Passagem de Parâmetro para Função

- Passagem de parâmetro é sempre por valor:
  - Entretanto, tipos mais "complexos" como table e function são armazenadas como referências!
- Caso necessite realizar passagem de parâmetro com tipos mais simples...
  - Usar retorno da função

## Passagem de Parâmetro para Função

- E agora? O que vai ser impresso na tela?

```
function soma (a)
  a = a + 2
end
(c) a = 2
    print (a)
    soma (a)
    print (a)
```

```
shell>$ lua soma.lua
2
2
shell>$
```

```
function soma ()
  a = a + 2
end
(d) a = 2
    print (a)
    soma ()
    print (a)
```

```
shell>$ lua soma.lua
2
4
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Passagem de Parâmetro para Função

- E agora? O que vai ser impresso na tela?

```
function soma (a)
  a = a + 2
end
(c) a = 2
    print (a)
    soma (a)
    print (a)
```

```
shell>$ lua soma.lua
2
4
2
Passagem de parâmetro por
valor
```

```
function soma ()
  a = a + 2
end
(d) a = 2
    print (a)
    soma ()
    print (a)
```

```
shell>$ lua soma.lua
2
4
Uso da variável criada
globalmente
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Passagem de Parâmetro para Função

- E agora? O que vai ser impresso na tela?

```
function soma (a)
  a = a + 2
end
(e) local a = 2
    print (a)
    soma (a)
    print (a)
```

```
shell>$ lua soma.lua
2
2
```

Passagem de parâmetro por valor

```
function soma ()
  a = a + 2
end
(f) local a = 2
    print (a)
    soma ()
    print (a)
```

```
shell>$ lua soma.lua
ERRO! a = nil...
shell>$
```

Variável criada localmente após a declaração da função

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## E se os parâmetros vierem do terminal?

```
print ("O programa recebeu ", #arg, " argumentos: ", arg[0], arg[1], arg[2], " e ", arg[3])
```

```
shell>$ lua paramTerminal.lua oi 10 a
O programa recebeu 3 argumentos: paramTerminal.lua, oi, 10 e a
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Bibliotecas Padrão

- As bibliotecas padrão de Lua oferecem funções úteis
  - São implementadas diretamente através da API C
    - Algumas dessas funções oferecem serviços essenciais para a linguagem (ex. type)
    - Outras oferecem acesso a serviços "externos" (ex. E/S)
  - Funções poderiam ser implementadas em Lua
    - Entretanto, são bastante úteis ou possuem requisitos de desempenho críticos que levam ao uso da implementação em C (ex. table.sort)

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Biblioteca de I/O

- Utilizada para operações de leitura e escrita
  - Função read
    - Pode receber um argumento que define o tipo de valor lido:
      - io.read("\*number") → Lê um número
      - io.read("\*line") → Lê a linha
  - Função write
    - Escreve um número arbitrário de strings passadas como argumento no stdout
      - io.write(var, "qualquer coisa")
        - A variável "var" também contém uma string

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Exemplo 1: Fatorial

- Escreva um programa em Lua para calcular o número fatorial de um inteiro passado pelo usuário



## Exemplo 1: Fatorial

```
function fatorial (n)
  if n == 1 then
    return 1
  else
    return n * fatorial (n-1)
  end
end

local n = io.read (**number*)
print ('Fatorial é:', fatorial (n))
```

## Exemplo 2: Fibonacci

- Escreva um programa em Lua para calcular o enésimo número da série de Fibonacci.
  - O enésimo número é passado pelo usuário



## Exemplo 2: Fibonacci

```
function fibonacci (n)
  if n == 0 then
    return 0
  elseif n == 1 then
    return 1
  else
    return fibonacci (n-2) + fibonacci (n-1)
  end
end

local n = io.read (**number*)
print ( 'Resultado', fibonacci (n))
```

## Exemplo 3: Soma de Matrizes

- Escreva um programa em Lua para calcular a soma de duas matrizes quadradas



```
function soma_matriz(t1, t2)
  local tr = {}
  for i=1, n do
    for j=1, n do
      tr [n*(i-1)+j] = t1 [n*(i-1)+j] + t2 [n*(i-1)+j]
    end
  end
  return tr
end

local A, B = {}, {}
-- Número de elementos da matriz
n = io.read (**number*)
-- Definição dos elementos das matrizes de entrada
for i=1, n do
  for j=1, n do
    A [n*(i-1)+j] = n*(i-1)+j
  end
end
for i=1, n do
  for j=1, n do
    table.insert(B, n*(n*(i-1)+j))
  end
end
for i=1, n do
  for j=1, n do
    print (A [n*(i-1)+j], B [n*(i-1)+j])
  end
end

-- Resultado da soma
local S = soma_matriz(A, B)
for i=1, n do
  for j=1, n do
    print (S[n*(i-1)+j])
  end
end
```

## Exemplo 4: Lista Encadeada

- Escreva um programa em Lua para inserir elementos em uma lista encadeada



## Exemplo 4: Lista Encadeada

```
function insert (listPar)
  l = {next = listPar, value = listPar.value + 1}
  return l
end

local list = {next = nil, value = 1}
for i = 1, 5 do
  list = insert (list)
end
for i = 1, 6 do
  print ("%v = ", list.value)
  list = list.next
end
```

## Linguagem Perl

- Criada em 1987 por Larry Wall na Unisys
  - Baseada em C, shell script, AWK e sed
- Linguagem de script dinâmica
  - Semelhante a Python, PHP e Ruby
- Possui simplicidade de codificação, eficiência e portabilidade
- Possui possibilidade de embutir o interpretador em uma aplicação C

## Linguagem Perl

- Possui propósito geral
  - Pode ser utilizada em...
    - Pequenos scripts e sistemas complexos
- Principais aplicações
  - Processamento de texto
    - Aplicação original
  - Web
    - Amazon.com, BBC Online, Ticketmaster
  - Desenvolvimento de software
    - Twiki
  - Comunicações

## Linguagem Perl

- Perl é uma linguagem interpretada
  - Interpretador: perl
    - Interpreta e executa o código
      - perl <arq-código>

## Primeiro Exemplo em Perl

- Programa: HelloWorld.pl

```
print "Hello, world!\n";
```

- Interpretação: perl HelloWorld.pl

```
shell->$ perl HelloWorld.pl
Hello, world!
shell->$
```

## Primeiro Exemplo em Perl

- Programa: HelloWorld.pl

```
print "Hello, world!\n";
```

- Interpretação: perl HelloWorld.pl

```
shell>$ perl HelloWorld.pl
Hello, world!
shell>$
```

Distribuição de Perl para Windows: "Strawberry Perl"  
<http://strawberryperl.com>

## Primeiro Exemplo em Perl

- Modo pela linha de comando:

```
shell>$ perl -e 'print "Hello, world!\n";'
Hello, world!
shell>$
```

- Modo interativo: Somente em modo debug...

```
shell>$ perl -de0
...
<DB 1> print "Hello, world!\n";
Hello, world!
<DB 2>
```

## Variáveis em Perl

- Escopo
  - Por padrão, as variáveis são sempre globais
    - Para indicar variáveis locais, usa-se a palavra-chave: `my`
- Tipos
  - Perl possui três tipos principais
    - scalar
      - Representa um único valor
    - array
      - Representa uma lista de valores
    - hash
      - Representa um conjunto de pares chave/valor

## Variáveis em Perl

- Escopo
  - Por padrão, as variáveis são sempre globais
    - Para indicar variáveis locais, usa-se a palavra-chave: `my`
- Tipos
  - Perl possui três tipos principais
    - scalar
      - Representa um único valor
    - array
      - Representa uma lista de valores
    - hash
      - Representa um conjunto de pares chave/valor

Podem ser strings, inteiros e pontos flutuantes. O Perl converte automaticamente entre os tipos. Essas variáveis devem ser sempre precedidas por `$`

## Variáveis em Perl

- Escopo
  - Por padrão, as variáveis são sempre globais
    - Para indicar variáveis locais, usa-se a palavra-chave: `my`
- Tipos
  - Perl possui três tipos principais
    - scalar
      - Representa um único valor
    - array
      - Representa uma lista de valores
    - hash
      - Representa um conjunto de pares chave/valor

Representa uma lista de valores que podem ser escalares. Essas variáveis devem ser sempre precedidas por `@`

## Variáveis em Perl

- Escopo
  - Por padrão, as variáveis são sempre globais
    - Para indicar variáveis locais, usa-se a palavra-chave: `my`
- Tipos
  - Perl possui três tipos principais
    - scalar
      - Representa um único valor
    - array
      - Representa uma lista de valores
    - hash
      - Representa um conjunto de pares chave/valor

Representa um conjunto de chaves e valores associados. Essas variáveis devem ser sempre precedidas por `%`

## Segundo Exemplo em Perl

- Programa: `escalares.pl`

```
my $s = 3; # Variável escalar receber um inteiro
print "A variável é: $s\n"; # Imprime 3
$s = "perl"; # Variável escalar receber uma string
print "A variável é: ", $s, " agora\n"; # Imprime perl
print "A variável é: $s agora\n";
$s = 1.23;
print $s+1, "\n"; # Imprime 2.23
```

## Segundo Exemplo em Perl

- Programa: `escalares.pl`

```
my $s = 3; # Variável escalar receber um inteiro
print "A variável é: $s\n"; # Imprime 3
$s = "perl"; # Variável escalar receber uma string
print "A variável é: ", $s, " agora\n"; # Imprime perl
print "A variável é: $s agora\n";
$s = 1.23;
print $s+1, "\n"; # Imprime 2.23
```

Variável escalar "s" é declarada local

## Segundo Exemplo em Perl

- Programa: `escalares.pl`

```
my $s = 3; # Variável escalar receber um inteiro
print "A variável é: $s\n"; # Imprime 3
$s = "perl"; # Variável escalar receber uma string
print "A variável é: ", $s, " agora\n"; # Imprime perl
print "A variável é: $s agora\n";
$s = 1.23;
print $s+1, "\n"; # Imprime 2.23
```

Marcação de término da sentença

## Segundo Exemplo em Perl

- Programa: `escalares.pl`

```
my $s = 3; # Variável escalar receber um inteiro
print "A variável é: $s\n"; # Imprime 3
$s = "perl"; # Variável escalar receber uma string
print "A variável é: ", $s, " agora\n"; # Imprime perl
print "A variável é: $s agora\n";
$s = 1.23;
print $s+1, "\n"; # Imprime 2.23
```

Comentário

## Segundo Exemplo em Perl

- Programa: `escalares.pl`

```
my $s = 3; # Variável escalar receber um inteiro
print "A variável é: $s\n"; # Imprime 3
$s = "perl"; # Variável escalar receber uma string
print "A variável é: ", $s, " agora\n"; # Imprime perl
print "A variável é: $s agora\n";
$s = 1.23;
print $s+1, "\n"; # Imprime 2.23
```

- Interpretação: `perl escalares.pl`

```
shell>$ perl escalares.pl
A variável é: 3
A variável é: perl agora
A variável é: perl agora
2.23
shell>$
```

## Segundo Exemplo em Perl

- Programa: `escalares.pl`

```
$s = 3; # Variável escalar receber um inteiro
print "A variável é: $s\n"; # Imprime 3
$s = "perl"; # Variável escalar receber uma string
print "A variável é: ", $s, " agora\n"; # Imprime perl
print "A variável é: $s agora\n";
$s = 1.23;
print $s+1, "\n"; # Imprime 2.23
```

Variável passa a ser global. Assim, como em Lua, essa opção não é tão eficiente e é evitada. O resultado de execução, porém, é o mesmo

## Segundo Exemplo em Perl

- Programa: `escalares.pl`

```
$$ = 3; # Variável escalar receber um inteiro
print "A variável é: $$\n"; # Imprime 3
$$ = 'perl'; # Variável escalar receber uma string
print "A variável é: ", $$, " agora\n"; # Imprime perl
print "A variável é: $$ agora\n";
$$ = 1.23;
print $$+1, "\n"; # Imprime 2.23
```

- Interpretação: `perl escalares.pl`

```
shell>$ perl escalares.pl
A variável é: 3
A variável é: perl agora
A variável é: perl agora
2.23
shell>$
```

## Terceiro Exemplo em Perl

- Programa: `vetores.pl`

```
my @animals = ('camel', 'llama', 'owl');
my @numbers = (23, 42, 69);
my @mixed = ('camel', 42, 1.23);

print $animals [0], "\n"; # Imprime 'camel'
print $animals [1], "\n"; # Imprime 'llama'
print $mixed [$#mixed], "\n"; # Imprime último elemento, imprime 1.23

print @animals [0..1], "\n"; # Imprime 'camel' e 'llama'
print @animals [0..2], "\n"; # Imprime tudo: 'camel', 'llama' e 'owl'
print @animals [!..$#animals], "\n"; # Imprime 'llama' e 'owl'
print @animals; # Imprime tudo
```

## Terceiro Exemplo em Perl

- Programa: `vetores.pl`

```
my @animals = ('camel', 'llama', 'owl');
my @numbers = (23, 42, 69);
my @mixed = ('camel', 42, 1.23);

print $animals [0], "\n"; # Imprime 'camel'
print $animals [1], "\n"; # Imprime 'llama'
print $mixed [$#mixed], "\n"; # Imprime último elemento

print @animals [0..1], "\n"; # Imprime 'camel' e 'llama'
print @animals [0..2], "\n"; # Imprime tudo: 'camel',
print @animals [!..$#animals], "\n"; # Imprime 'llama'
print @animals; # Imprime tudo
```

- Interpretação:  
`perl vetores.pl`

```
shell>$ perl vetores.pl
camel
llama
1.23
camelllama
camelllamaowl
llamaowl
camelllamaowl
shell>$
```

## Quarto Exemplo em Perl

- Programa: `hashes.pl`

```
my %cores_frutas = ('maca', 'vermelha', 'banana', 'amarela');

print $cores_frutas{'maca'}, "\n";

%cores_frutas = (
    maca => 'verde',
    banana => 'preta',
);

print $cores_frutas{'maca'}, "\n";

my @frutas = keys %cores_frutas;
my @cores = values %cores_frutas;

print @frutas, "\n", @cores;
```

- Interpretação:  
`perl hashes.pl`

## Quarto Exemplo em Perl

- Programa: `hashes.pl`

```
my %cores_frutas = ('maca', 'vermelha', 'banana', 'amarela');

print $cores_frutas{'maca'}, "\n";

%cores_frutas = (
    maca => 'verde',
    banana => 'preta',
);

print $cores_frutas{'maca'}, "\n";

my @frutas = keys %cores_frutas;
my @cores = values %cores_frutas;

print @frutas, "\n", @cores;
```

- Interpretação:  
`perl hashes.pl`

```
shell>$ perl hashes.pl
vermelha
verde
macabanana
verdepreta
shell>$
```

## Quinto Exemplo em Perl

- Programa: `hashHashes.pl`

- Tipos mais complexos de dados podem ser construídos usando referências

- Referências são variáveis escalares

```
# Variável escalar que recebe uma referência
my $variables = {
    scalar => {
        description => "single item",
        sigil => '$',
    },
    array => {
        description => "ordered list of items",
        sigil => '@',
    },
    hash => {
        description => "key/value pairs",
        sigil => '%',
    },
};

print "Scalars begin with a $variables->{'scalar'}->{'sigil'}\n";
```



## Quinto Exemplo em Perl

- Programa: `hashHashes.pl`
  - Tipos mais complexos de dados podem ser construídos usando referências

- Referências são variáveis escalares

# Variável escalar que recebe uma referência

```
my $variables = {
    scalar => {
        description => 'single item',
        sigil => '$',
    },
    hash => {
        description => 'key/value pairs',
        sigil => '%',
    },
};

print "Scalars begin with a $variables->{'scalar'}->{'sigil'}\n";
```

```
shell-> perl hashHashes.pl
Scalars begin with a $
shell->
```

## Referências

- Duas maneiras para obter as referências: `\` ou `[]`

`$aref = \@array;` # `$aref` é uma referência para `@array`

`$href = \%hash;` # `$href` é uma referência para `%hash`

`$sref = \$scalar;` # `$sref` é uma referência para `$scalar`

`$aref = [ 1, "foo", nil, 13 ];`

# `$aref` é uma referência para um array

`$href = { APR => 4, AUG => 8 };`

# `$href` é uma referência para um hash

**Na segunda maneira, a variável foi criada diretamente como uma referência**

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Exemplo Usando Referências

- Programa: `refs.pl`

```
# Isso
my @arrayref = [ 1, 2, 3 ];
$aref = [ 1, 2, 3 ];
print ${$aref}[1], " ", $aref->[1], "\n";

# É análogo disso
@array = (4, 5, 6);
$aref = \@array;
print $array [1], " ", ${$aref}[1], " ", $aref->[1], "\n";
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Exemplo Usando Referências

- Programa: `refs.pl`

```
# Isso
my @arrayref = [ 1, 2, [ 'a', 'b', 'c' ] ];
$aref = [ 1, 2, 3 ];
print ${$aref}[1], " ", $aref->[1], "\n";

# É análogo disso
@array = (4, 5, 6);
$aref = \@array;
print $array [1], " ", ${$aref}[1], " ", $aref->[1], "\n";
```

- Interpretação:

```
perl refs.pl
```

```
shell-> perl refs.pl
2 2
5 5
shell->
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Mais Um Exemplo Usando Referências

```
# Referência para um array
my @arrayref = [ 1, 2, [ 'a', 'b', 'c' ] ];
print $arrayref->[2][1], "\n";
print ${$arrayref}[2][1], "\n\n";

my $a = 'exemplo';
my @b = (1, 2);
my %c = {
    nota1 => 5,
    nota2 => 7,
};

# Lista de referências
my @list = (\$a, \@b, \%c);
print ${$list}[0], "\n";
print $list[1]->[1], "\n", ${$list}[1][1], "\n\n";
print $list[2]->{nota1}, "\n", ${$list}[2]{nota1}, "\n\n";

# Equivalente à lista de referências
my @listRef = (\$a, \@b, \%c);
print ${$listRef}[0], "\n";
print $listRef[1]->[1], "\n", ${$listRef}[1][1], "\n\n";
print $listRef[2]->{nota1}, "\n", ${$listRef}[2]{nota1}, "\n\n";

# Atribuição
$listRef[2]->{nota1} = 8;
print $listRef[2]->{nota1}, "\n", ${$listRef}[2]{nota1}, "\n\n";

# Desreferenciação do array
my @l = @{$listRef[1]};
print $l[1], "\n\n";

# Desreferenciação do hash
my %h = %{$listRef[2]};
print $h{nota2}, "\n\n";
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Mais Um Exemplo Usando Referências

```
# Referência para um array
my @arrayref = [ 1, 2, [ 'a', 'b', 'c' ] ];
print $arrayref->[2][1], "\n";
print ${$arrayref}[2][1], "\n\n";

my $a = 'exemplo';
my @b = (1, 2);
my %c = {
    nota1 => 5,
    nota2 => 7,
};

# Lista de referências
my @list = (\$a, \@b, \%c);
print ${$list}[0], "\n";
print $list[1]->[1], "\n", ${$list}[1][1], "\n\n";
print $list[2]->{nota1}, "\n", ${$list}[2]{nota1}, "\n\n";

# Equivalente à lista de referências
my @listRef = (\$a, \@b, \%c);
print ${$listRef}[0], "\n";
print $listRef[1]->[1], "\n", ${$listRef}[1][1], "\n\n";
print $listRef[2]->{nota1}, "\n", ${$listRef}[2]{nota1}, "\n\n";

# Atribuição
$listRef[2]->{nota1} = 8;
print $listRef[2]->{nota1}, "\n", ${$listRef}[2]{nota1}, "\n\n";

# Desreferenciação do array
my @l = @{$listRef[1]};
print $l[1], "\n\n";

# Desreferenciação do hash
my %h = %{$listRef[2]};
print $h{nota2}, "\n\n";
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

```
shell-> perl refs2.pl
b
b
exemplo
2
2
5
5
exemplo
2
2
8
2
7
shell->
```

## Sexto Exemplo em Perl

```
my %hashed = ("maca", "verde", "banana", "amarela");
my @values = values %hashed;
print @values, "\n";

# Maneira equivalente para atribuir valores na hash
# Uso do operador =>
%hashed = (
    maca => "madura",
    banana => "estragada",
);

my @new_values = values %hashed;
print @new_values, "\n";

my @sorted_values = sort @new_values;
print @sorted_values, "\n";
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Sexto Exemplo em Perl

```
my %hashed = ("maca", "verde", "banana", "amarela");
my @values = values %hashed;
print @values, "\n";

# Maneira equivalente para atribuir valores na hash
# Uso do operador =>
%hashed = (
    maca => "madura",
    banana => "estragada",
);

my @new_values = values %hashed;
print @new_values, "\n";

my @sorted_values = sort @new_values;
print @sorted_values, "\n";
```

```
shell>$ perl ordena.pl
verdeamarela
maduraestragada
estragadamadura
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Operadores em Perl

- Relacionais numéricos
  - <, >, <=, >=, ==, !=
- Relacionais strings
  - eq, ne, lt, gt, le, ge
- Lógicos
  - && (and), || (or), ! (not)

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Estruturas de Controle em Perl

- Condição:

```
if ($x > 10) {
    print "x > 10";
} elsif ($x > 5) {
    print "5 > x > 10";
} else {
    print "x < 5";
}
```

```
# precisa das chaves mesmo se houver apenas uma linha no bloco
unless ($x == 10) {
    print "x != 10";
} # Mesmo que
if ($x != 10) {...}
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Estruturas de Controle em Perl

- Laço:

```
while ($x < 10) {
    $x = $x + 1;
}
```

```
# valor inicial, cond. de contorno e passo
for ($x=1, $x<10, $x++) {
    print $x;
}
# Varre o vetor
foreach (@vetor) {
    print "Elemento $_"; # $_ var. padrão
}
# Chaves %hash
foreach (keys %hash) {
    print "Chaves $_";
}
# Sem a var. padrão
foreach my $k (keys %hash) {
    print "Chaves $k";
}
```

## Estruturas de Controle em Perl

```
my $value = 1;

# modo tradicional
if ($value) {
    print "Verdade!\n";
}
# Modo de pós-condição do Perl
print "Verdade!\n\n" if $value;

my @array = (1, 2, 3);

foreach (@array) {
    print "Elemento é $_\n";
}
print "Pós-cond: $array[$_]\n" foreach 0 .. $#array;
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Estruturas de Controle em Perl

```

my $value = 1;

# modo tradicional
if ($value) {
    print "Verdade!\n";
}

# Modo de pós-condição do Perl
print "Verdade!\n" if $value;

my @array = (1, 2, 3);

foreach (@array) {
    print "Elemento é $_\n";
}

print "Pós-cond: $array[$_]\n" foreach 0 .. $#array;
    
```

Obriga uso das chaves

Dispensa uso das chaves

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Sub-rotinas em Perl

- Declaração de sub-rotinas

– sub

```

sub nome-da-funcao {
    corpo da função;
}
    
```

• Ex: sub fatorial(n)

- Funções podem receber e retornar n parâmetros

```

sub nome-da-funcao {
    ($par1, $par2, ..., $parn) = @_;
    corpo da função;
    return $par1, $par2, ..., $parn;
}

($x1, $x2, ..., $xn) = nome-da-funcao(arg1, arg2, ..., argn);
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Sétimo Exemplo em Perl

```

sub ordena {
    ($v1, $v2, $v3) = sort @_;
    return ($v1, $v2, $v3);
}

my @v = (3, 2, 1);

print ordena (@v), "\n";
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Sétimo Exemplo em Perl

```

sub ordena {
    ($v1, $v2, $v3) = sort @_;
    return ($v1, $v2, $v3);
}

my @v = (3, 2, 1);

print ordena (@v), "\n";
    
```

```

shell>$ perl ordenaSub.pl
123
shell>$
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Passagem de Parâmetro no Perl

- Pode ser por valor ou referência

– Usando Escalares

```

sub retornaEscalar {
    my $r = $_[0];
    print $r, "\t", $_[0], "\n";
    $r++;
    print $r, "\t", $_[0], "\n";
    return $r;
}

sub retornaEscalarRef {
    my $r = $_[0];
    print $r, "\t", $($r), "\t", $($_[0]), "\n";
    $($r)++;
    print $r, "\t", $($r), "\t", $($_[0]), "\n";
    return $r; # Poderia retornar também $_[0]
}

my $v = 3;
my $escalar = retornaEscalar($v);
print "Escalar Retornado = ", $escalar, "\tEscalarValor = ", $v, "\n";

my $ref = 3;
my $escalarRef = retornaEscalarRef(\$ref);
print "Ref = ", $ref, "\tEscalarRef = ", $escalarRef, "\n";
    
```

```

shell>$ perl funcoes.pl
3      3
4      3
Escalar Retornado = 4      Escalar Valor = 3
SCALAR(0x1870ea0) 3      3
SCALAR(0x1870ea0) 4      4
Ref = 4      EscalarRef = 4
shell>$
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Passagem de Parâmetro no Perl

- Pode ser por valor ou referência

– Usando Arrays

```

sub retornaArray {
    my @r = @_;
    print @r, "\t", $_[0], "\t", $_[1], "\n";
    $@[0]++; $@[1]++;
    print @r, "\t", $_[0], "\t", $_[1], "\n";
    return @r;
}

sub retornaArrayRef {
    my $r = $_[0]; # Referência é escalar
    print $r, "\t", $($r), "\t", $($_[0]), "\n";
    $($r)[0]++; $($r)[1]++;
    print $r, "\t", $($r), "\t", $($_[0]), "\t", $($_[1]), "\n";
    return $r; # Poderia retornar também $_[0]
}

my @v = (2, 3);
my @array = retornaArray(@v);
print "Array Retornado = ", @array, "\tArrayValor = ", @v, "\n";

my $ref = (4, 5);
my @arrayRef = retornaArrayRef(\$ref);
print "Ref = ", $ref, "\tArrayRef = ", @arrayRef, "\n";
    
```

```

shell>$ perl funcoesArray.pl
23     2     3
34     2     3
Array Retornado = 34 Array Valor = 23
ARRAY(0x1870ea0) 45     4     5
ARRAY(0x1870ea0) 56     5     6
Ref = 56      ArrayRef = 56
shell>$
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Passagem de Parâmetro no Perl

- Pode ser por valor ou ref
- Usando Hashes

```
shell->$ perl funcoesHash.pl
y3x2 y3x2 2 3
y4x3 y3x2 3 4
Hash Retornado = y4x3 Hash Valor = y3x2
HASH(0x1870aad) y3x4 4 5
HASH(0x1870aad) y6x5 5 6
Ref = y6x5 HashRef = y6x5
shell->$
```

```
sub retornaHash {
    my %s = @_; # Hashes são passadas como
                # um array de elementos chave-valor
    print %s, "\n";
    $s{"a"}++; $s{"b"}++;
    print %s, "\n";
    return %s;
}

sub retornaHashRef {
    my $r = $_[0]; # Referência é escalar
    print $r, "\n";
    $r{"a"}++; $r{"b"}++;
    print $r, "\n";
    return $r; # Poderia retornar também $_[0]
}

my %s = (x => 2, y => 3);
my %hash = retornaHash(%s);
print "Hash Retornado = ", %hash, "\n";

my %ref = (x => 4, y => 5);
my %hashRef = retornaHashRef(\%ref);
print "Ref = ", %hashRef, "\n";
```

Prof. Miguel Campista

## Passagem de Parâmetro no Perl

- Pode ser por valor ou referência
- Usando múltiplos argumentos... Usar referências!

```
use warnings;
use strict;

sub imprimeArgs {
    print $_[0], "\n";
    print $_[1], "\n";
    print $_[2], "\n";
    print $_[3], "\n";
    print $_[4], "\n";
    print $_[5], "\n";
}

sub imprimePorRefs {
    my $ref = $_[0]; my $ref2 = $_[1]; my $ref3 = $_[2];
    # Referência é escalar
    print @$ref, "\n";
}

my $a = 1; my @a = (2, 3); my %h = (x => 4, y => 5);
imprimeArgs($a, @a, %h);
imprimePorRefs(\%a, \@a, \%h);
```

```
shell->$ perl funcoesConcatenadas.pl
1 2 3 y 5 x 4
1 23 y5x4
shell->$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## E se os parâmetros vierem do terminal?

```
print "O programa recebeu ", $#ARGV + 1, " argumentos: @ARGV[0], @ARGV[1] e @ARGV[2] \n";
```

```
shell->$ perl paramTerminal.pl oi 10 a
O programa recebeu 3 argumentos: oi, 10 e a
shell->$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Entrada e Saída

- Uso das funções `open ()` e `close ()`
- Abertura e fechamento de arquivos, respectivamente

```
# Entrada
open (my $in, "<", "input.txt") or die "Can't open input.txt: $!";

#Saída
open (my $in, ">", "output.txt") or die "Can't open output.txt: $!";

#Concatenação
open (my $in, ">>", "log.txt") or die "Can't open log.txt: $!";

#Fechamento
close $in or die "$in: $!";
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Oitavo Exemplo em Perl

```
# O $! imprime o motivo...
open (my $in, "<", "input.txt") or die "Can't open input.txt: $!";

while (<$in>) {
    print $_;
}

# Volta para a primeira linha do arquivo
seek ($in, 0, 0);

print "**** Só a primeira linha ****\n";
my $line = <$in>;
print $line;

# Novamente, volta para a primeira linha do arquivo
seek ($in, 0, 0);

print "**** Arquivo todo ****\n";
my @lines = <$in>;
print @lines;

close $in or die "Can't close input.txt: $!";
```

Arquivo: input.txt  
Exemplo  
Entrada  
Saída  
Usando  
Arquivos

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Oitavo Exemplo em Perl

```
# O $! imprime o motivo...
open (my $in, "<", "input.txt") or die "Can't open input.txt: $!";

while (<$in>) {
    print $_;
}

# Volta para a primeira linha do arquivo
seek ($in, 0, 0);

print "**** Só a primeira linha ****\n";
my $line = <$in>;
print $line;

# Novamente, volta para a primeira linha do arquivo
seek ($in, 0, 0);

print "**** Arquivo todo ****\n";
my @lines = <$in>;
print @lines;

close $in or die "Can't close input.txt: $!";
```

```
shell->$ perl arquivoES.pl
Exemplo
Entrada
Saída
Usando
Arquivos
**** Só a primeira linha ****
Exemplo
**** Arquivo todo ****
Exemplo
Entrada
Saída
Usando
Arquivos
shell->$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Oitavo Exemplo em Perl

```
# O $! imprime o motivo...
open (my $in, "<" , "input.txt") or die "Can't open input.txt: $!";

while (<$in> {
    print $_;
}

# Volta para a primeira linha do arquivo
seek ($in, 0, 0);

print "====> Primeira linha ***\n";
my $line = <$in>;
print "$line\n";

# Novamente, volta para a primeira linha do arquivo
seek ($in, 0, 0);

print "====> Todos ***\n";
my @lines = <$in>;
print @lines;

close $in or die "Can't close input.txt: $!";
```

Operador <> lê apenas uma linha em contexto de escalar

Operador <> lê o arquivo todo em contexto de array

\$ perl arquivoES.pl  
Entrada  
Saída  
Usando  
Arquivos  
Exemplo  
Entrada  
Saída  
Usando  
Arquivos  
shell>

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Módulos

- A linguagem Perl é rica em módulos
  - Carregados com o uso do use

```
my @array = (1, 2, 3);
my %hash = ('x', 4, 'y', 5, 'z', 6);

@array = @hash;
print values %hash, "\n";
```

```
shell>$ perl modulo.pl
546
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Módulos

- A linguagem Perl é rica em módulos
  - Carregados com o uso do use

```
# Apresenta warnings durante execução caso haja potenciais problemas
use warnings;
```

```
my @array = (1, 2, 3);
my %hash = ('x', 4, 'y', 5, 'z', 6);

@array = @hash;
print values %hash, "\n";
```

```
shell>$ perl modulo.pl
Name "main::hash" used only once: possible typo at modulo.pl line 9.
546
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Módulos

- A linguagem Perl é rica em módulos
  - Carregados com o uso do use

```
# Apresenta warnings durante execução caso haja potenciais problemas
use warnings;
# Para a execução caso haja potenciais problemas
use strict;
```

```
my @array = (1, 2, 3);
my %hash = ('x', 4, 'y', 5, 'z', 6);

@array = @hash;
print values %hash, "\n";
```

```
shell>$ perl modulo.pl
Global symbol "@hash" requires explicit package name at modulo.pl line 9.
Execution of modulo.pl aborted due to compilation errors.
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Módulos

- A linguagem Perl é rica em módulos
  - Carregados com o uso do use

```
use warnings;

$DEBUG = 1;

print $DEBUG, "\n";
```

```
miguel@pegasus-linux:~$ perl global-strict.pl
1
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Módulos

- A linguagem Perl é rica em módulos
  - Carregados com o uso do use

```
use warnings;

$DEBUG = 1;

print $DEBUG, "\n";
```

```
miguel@pegasus-linux:~$ perl global-strict.pl
1
```

Uma coisa interessante do módulo strict é que ele não permite o uso de variáveis globais.

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Módulos

- A linguagem Perl é rica em módulos
  - Carregados com o uso do `use`

```
use warnings;
use strict;

$DEBUG = 1;

print $DEBUG, "\n";
```

```
miguel@pegasus-linux:~$ perl global-strict.pl
Global symbol "$DEBUG" requires explicit package name at global-strict.pl line 4
Global symbol "$DEBUG" requires explicit package name at global-strict.pl line 6
Execution of global-strict.pl aborted due to compilation errors.
```

## Módulos

- A linguagem Perl é rica em módulos
  - Carregados com o uso do `use`

O `our` define que a variável `DEBUG` foi definida em outro escopo e que pode ser usada no escopo atual

```
use warnings;
use strict;

our $DEBUG = 1;

print $DEBUG, "\n";
```

```
miguel@pegasus-linux:~$ perl global-strict.pl
1
```

## Exemplo 1: Fatorial

- Escreva um programa em Perl para calcular o número fatorial de um inteiro passado pelo usuário

?

## Exemplo 1: Fatorial

```
sub fatorial {
    my $n = shift;
    if ($n == 1) {
        return 1;
    } else {
        return $n*fatorial($n-1);
    }
}

my $n = <STDIN>;
print fatorial($n), "\n";
```

## Exemplo 2: Fibonacci

- Escreva um programa em Perl para calcular o *n*ésimo número da série de Fibonacci.
  - O *n*ésimo número é passado pelo usuário

?

## Exemplo 2: Fibonacci

```
sub fibonacci {
    my ($n) = @_;
    if ($n == 0) {
        return 0;
    } elsif ($n == 1) {
        return 1;
    } else {
        return fibonacci($n-1) + fibonacci($n-2);
    }
}

my $n = <STDIN>;
print fibonacci($n), "\n";
```

## Exemplo 3: Ordenamento de Cadastro

- Escreva um programa em Perl que ordene os nomes passados por um usuário



```
use warnings;

sub leitura {
    my $n = shift;
    my @cad;
    for (my $i = 0; $i < $n; $i++) {
        print "Entre com o nome #<math>i</math>:\n";
        my $name = <STDIN>;
        chop($name);
        @cad = (@cad, $name);
    }
    return @cad;
}

sub imprime {
    foreach (@_) {
        print $_, "\n";
    }
}

print "Entre com o numero de cadastros:\n";
# Leitura do teclado do NUMERO de cadastros
my $n = <STDIN>;

# Função para leitura
my @cad = leitura($n);

# Função para impressão da lista
print "Lista desordenada!\n";
imprime(@cad);

# Uso da função sort
print "Lista Ordenada!\n";
imprime(sort(@cad));
```

## Exemplo 4: Ordenamento de Números

- E se ao invés de nomes, os elementos do array fossem números decimais



## Expressões Regulares

- Avaliação de presença de expressões regulares
  - Simples "match"
  - Se a variável for \$\_  
`if (/foo/) {...}`
  - Caso contrário...  
`if ($s =~ /foo/) {...}`

**Operador =~ usado para comparar uma expressão escalar com um padrão**

## Expressões Regulares

- Avaliação de presença de expressões regulares
  - Substituição simples

`$s =~ s/foo/bug/`

**Muitas outras podem ser vistas na documentação do Perl...**

## Expressões Regulares

```
my $s = <STDIN>;
print "Nome inserido: ", $s;

print "Você buscar 'gu'\n";
if ($s =~ /gu/) {
    print "Achei gu\n";
    $s = s/gu/GUJ;
    print "Mudei para ", $s;
} else {
    print "Não achei 'gu'\n";
}

print "\n";
if ($s =~ /(S+)\s(S+)/) {
    print "Primeiro nome: $1\n";
    print "Segundo nome: $2\n";
} else {
    print "Nome: $s\n";
}
```

**Arquivo: expReg.pl**

```
miguel@pegasus-linux:~$ perl expReg.pl
miguel
Nome inserido: miguel
Você buscar "gu"
Achei gu
Mudei para miguel

Nome: miguel

miguel@pegasus-linux:~$ perl expReg.pl
miguel elias
Nome inserido: miguel elias
Você buscar "gu"
Achei gu
Mudei para miguel elias

Primeiro nome: miguel
Segundo nome: elias
```

## Expressões Regulares

```

my $S é caractere diferente de espaço em branco e \s
pr caractere de espaço em branco

print "Você buscar 'gu'\n";
if ($s =~ /gu/) {
    print "Achei gu\n";
    $$ = s/gu/GU/;
    print "Mudei para ", $$;
} else {
    print "Não achei 'gu'\n";
}

print "\n";
if ($s =~ /(\S+)\s(\S+)/) {
    print "Primeiro nome: $1\n";
    print "Segundo nome: $2\n";
} else {
    print "Nome: $$\n";
}
    
```

```

miguel@pegasus-linux:~$ perl expReg.pl
miguel
Nome inserido: miguel
Você buscar "gu"
Achei gu
Mudei para miguel

Nome: miguel

miguel@pegasus-linux:~$ perl expReg.pl
miguel elias
Nome inserido: miguel elias
Você buscar "gu"
Achei gu
Mudei para miguel elias

Primeiro nome: miguel
Segundo nome: elias
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Criação de um Módulo em Perl

- Uso do programa: **h2xs**
  - Vem com a distribuição do Perl
    - Cria arquivo de extensões para o Perl (\*.xs) de cabeçalhos .h do C
  - Execução do programa cria estrutura de diretórios com:
    - Changes
      - Registra mudanças
    - Makefile.PL
      - Arquivo usado para gerar o Makefile
    - README
    - Diretório t
      - Arquivos para teste
    - Diretório lib
      - Arquivo do módulo

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Criação de um Módulo em Perl

- Uso do programa: **h2xs**
  - Opção -n: Nome do módulo

```

shell>$ h2xs -n testeModule
Writing testeModule/ppport.h
Writing testeModule/lib/testeModule.pm
Writing testeModule/testeModule.xs
Writing testeModule/fallback/const-c.inc
Writing testeModule/fallback/const-xs.inc
Writing testeModule/Makefile.PL
Writing testeModule/README
Writing testeModule/testeModule.t
Writing testeModule/Changes
Writing testeModule/MANIFEST
shell>$
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Criação de um Módulo em Perl

- Criação do módulo
  - Edição do arquivo \*.pm no diretório lib
    - Inserção da interface a ser exportada
    - Inserção da função
- Instalação
  - Criação do Makefile
  - Compilação
  - Cópia dos arquivos compilados para os diretórios padrão

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Criação de um Módulo em Perl

```

# This allows declaration use testeModule ':all';
# If you do not need this, moving things directly into @EXPORT or @EXPORT_OK
# will save memory.
our %EXPORT_TAGS = ( 'all' => [ qw(
) ] );

our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );

our @EXPORT = qw(
    oi
);

our $VERSION = '0.01';

sub oi {
    print shift;
}

# Preloaded methods go here.

;
__END__
# Below is stub documentation for your module. You'd better edit it!
=head1 NAME
    
```



Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Nono Exemplo em Perl

```

use testeModule;

oi ("Hello World!\n");
    
```

```

shell>$ perl oi.pl
Hello World!
shell>$
    
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista



## Criação de um Módulo em Perl

```
# This allows declaration use testeModule ':all';
# If you do not need this, moving things directly into @EXPORT or @EXPORT_OK
# will save memory.
our %EXPORT_TAGS = ( 'all' => [ qw(
) ] );
our @EXPORT_OK = qw( oi );
our @EXPORT = qw(
);
our $VERSION = '0.01';
sub oi {
    print shift;
}
# Preloaded methods go here.
;
__END__
# Below is stub documentation for your module. You'd better edit it!
=head1 NAME
```

Interface mais restritiva.  
Evita acessos errados

Criação da função oi

função

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Nono Exemplo em Perl

```
use testeModule qw( oi );
oi ("Hello World!\n");
```

```
shell>$ perl oi.pl
Hello World!
shell>$
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Criação de um Módulo em Perl

- Instalação: **COM** permissão de super usuário

```
shell>$ h2xs -n testeModule
...
shell>$ cd testeModule
shell/testeModule>$ perl Makefile.PL
shell/testeModule>$ make
shell/testeModule>$ sudo make install
```

- Instalação: **SEM** permissão de super usuário

```
shell>$ h2xs -n testeModule
...
shell>$ cd testeModule
shell/testeModule>$ perl Makefile.PL INSTALL_BASE=/home/mydir
shell/testeModule>$ make
shell/testeModule>$ make install
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Criação de um Módulo em Perl

- Instalação: **COM** permissão de super usuário

```
shell>$ h2xs -n testeModule
```

**SEM** permissão de super usuário requer a configuração da variável de ambiente `PERL5LIB` para que ela encontre o módulo no diretório escolhido. Para isso, mas um passo é necessário:

```
shell/testeModule>$ export PERL5LIB=/home/mydir/lib/perl5
```

```
shell>$ h2xs -n testeModule
```

```
...
shell>$ cd testeModule
shell/testeModule>$ perl Makefile.PL INSTALL_BASE=/home/mydir
shell/testeModule>$ make
shell/testeModule>$ make install
```

## Criação de um Módulo em Perl

- Instalação: **SEM** permissão de super usuário
  - Opção `-X`: Especifica que o módulo não está ligado com código em C

```
shell>$ h2xs -X -n testeModule
...
shell>$ cp testeModule/lib/testeModule.pm .
...
Incluir as subrotinas em testeModule.pm e apagar a linha:
    use AutoLoader qw(AUTOLOAD)
...
shell>$ chmod -R 777 testeModule
...
O diretório testeModule pode ser apagado...
shell>$ rm -rf testeModule
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Uso do Perl em um Código C/C++

- Implica incluir o interpretador Perl no código do programa C/C++
  - Ligação com a biblioteca Perl
    - Deve estar de acordo com os requisitos do programa C/C++
      - Ex.: Não se deve usar o interpretador como uma thread separada se o programa é executado em uma thread única
- Criação de uma instância do interpretador Perl
  - Invoca o interpretador para a execução do código em Perl
  - Após o uso do interpretador, ele deve ser destruído

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Configuração para Uso do Interpretador Perl

- `perl -V::cc:`
  - Verifica o compilador de C
- `perl -V::ld:`
  - Verifica o ligador
- `perl -MExtUtils::Embed -e ccopts`
  - Verifica os includes necessários
- `perl -MExtUtils::Embed -e ldopts`
  - Verifica as bibliotecas necessárias

Informações necessárias para compilar códigos com interpretador Perl. O próprio interpretador já oferece as informações necessárias

## Configuração para Uso do Interpretador Perl

```
CC=$(shell perl -V::cc)
CFLAGS=$(shell perl -MExtUtils::Embed -e ccopts)
LD=$(shell perl -V::ld)
LDFLAGS=$(shell perl -MExtUtils::Embed -e ldopts)

VERSION=0.1
all: perl-ex$(VERSION)

perl-ex$(VERSION).o: perl-ex$(VERSION).c
$(CC) $(CFLAGS) -o $@ -c $<

perl-ex$(VERSION): perl-ex$(VERSION).o
$(LD) -o $@ $? $(LDFLAGS)

clean:
rm -f perl-ex$(VERSION)
```

## Configuração para Uso do Interpretador Perl

```
CC=$(shell perl -V::cc)
CFLAGS=$(shell perl -MExtUtils::Embed -e ccopts)
LD=$(shell perl -V::ld)
LDFLAGS=$(shell perl -MExtUtils::Embed -e ldopts)
```

Pode usar direto o nome do compilador e ligador. Por exemplo, o g++. Assim:  
**CC=g++**  
**LD=g++**

```
VERSION=0.1
all: perl-ex$(VERSION)

perl-ex$(VERSION).o: perl-ex$(VERSION).c
$(CC) $(CFLAGS) -o $@ -c $<

perl-ex$(VERSION): perl-ex$(VERSION).o
$(LD) -o $@ $? $(LDFLAGS)

clean:
rm -f perl-ex$(VERSION)
```

## Inserção de Trecho de Código Perl

```
#include <EXTERN.h> /* from the Perl distribution */
#include <perl.h> /* from the Perl distribution */

PerlInterpreter *my_perl; /* The Perl interpreter */

int main(int argc, char **argv, char **env) {
    /* inicialização */
    PERL_SYS_INIT3(&argc, &argv, &env);

    /* criação de um interpretador */
    my_perl = perl_alloc();
    perl_construct(my_perl);
    PERL_EXIT_FLAGS |= PERL_EXIT_DESTRUCT_END;

    /* invocação do perl com argumentos */
    int perl_argc = 0;
    char *code = "print scalar (localtime).\"\\n\"";
    char *perl_argv[] = {argv[0], ""};
    perl_parse(my_perl, NULL, perl_argc, perl_argv, NULL);
    perl_run(my_perl);

    /* limpeza */
    perl_destruct(my_perl);
    perl_free(my_perl);

    /* término */
    PERL_SYS_TERM();

    return 0;
}
```

## Inserção de Trecho de Código Perl

```
miguel@pegasus-linux:~$ make -f Makefile.perl
'cc' -D REENTRANT -D GNU_SOURCE -D EBELIAN -fno-strict-aliasing -pipe -I/usr/local/include -D LARGEFILE_SOURCE -D FILE_OFFSET_BITS=64 -I/usr/lib/perl/5.10/CORE -o perl-ex01.o -c perl-ex01.c
'cc' -Wl,-E -L/usr/local/lib -L/usr/lib/perl/5.10/CORE -lperl -ldl -lm -lpthread -lc -lcrypt perl-ex01.o -o perl-ex01
miguel@pegasus-linux:~$ ./perl-ex01
Tue Mar 22 18:10:16 2011
```

## Funções e Macros

- **PERL\_SYS\_INIT3 e PERL\_SYS\_TERM**
  - Macros para inicializar e finalizar, respectivamente, tarefas necessárias para criar e remover o interpretador Perl em um código C
  - Só devem ser utilizados uma vez, independente do número de interpretadores utilizados
- **perl\_alloc, perl\_construct, perl\_destruct e perl\_free**
  - Funções usadas para criar e destruir um único interpretador

## Funções e Macros

- **PL\_EXIT\_DESTRUCT\_END** e **PL\_exit\_flags**
  - Flags necessárias para que o interpretador execute o bloco de término
- **perl\_parse**
  - Configura o interpretador usando opções de linhas de comando

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Chamada de Sub-rotinas Individuais

```
#include <EXTERN.h> /* from the Perl distribution */
#include <perl.h> /* from the Perl distribution */

PerlInterpreter *my_perl; /* The Perl interpreter */

int main(int argc, char **argv, char **env) {
    /* inicialização */
    char *arglist[] = {NULL};
    PERL_SYS_INT3(gargc, gargv, genv);

    /* criação de um interpretador */
    my_perl = perl_alloc();
    perl_construct(my_perl);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /* invocação do perl com argumentos */
    perl_parse(my_perl, NULL, argc, argv, NULL);
    call_argv("showtime", G_DISCARD | G_NOARGS, args);

    /* limpeza */
    perl_destruct(my_perl);
    perl_free(my_perl);

    /* término */
    PERL_SYS_TERM();

    return 0;
}
```

showtime.pl

```
print "I cant be printed.";
sub showtime {
    print time, "\n";
}
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Chamada de Sub-rotinas Individuais

```
miguel@pegasus-linux:~$ make -f Makefile.perl
'cc' -D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -pipe -I/usr/local/include -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -I/usr/lib/perl/5.10/CORE -o perl-ex02.o -c perl-ex02.c
'cc' -Wl,-E -L/usr/local/lib -L/usr/lib/perl/5.10/CORE -lperl -ldl -lm -lpthread -lc -lcrypt perl-ex02.o -o perl-ex02
miguel@pegasus-linux:~$ ./perl-ex02 showtime.pl
1300831965
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Chamada de Sub-rotinas Individuais

- Uso das funções **call\_\***
- **G\_NOARGS** e **G\_DISCARD**
  - Usadas quando a sub-rotina em Perl não possui nem argumentos nem valor de retorno, respectivamente
- **args**
  - Lista de argumentos a ser passada para as rotinas individuais
    - Lista de strings terminadas por **NULL**

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C/C++

```
#include <EXTERN.h>
#include <perl.h>

PerlInterpreter *my_perl;

main (int argc, char **argv, char **env) {
    char *embedding[] = {"", "c", "p", "g"};

    PERL_SYS_INT3(gargc, gargv, genv);
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, 0, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_run(my_perl);

    /** Trata o $a como um inteiro **/
    eval_gv["$a = 9; $a **= 2", TRUE];
    printf("a = %d\n", SvIV(get_sv("a", 0)));
    /** Trata o $a como um float **/
    eval_gv["$a = 3.14; $a **= 2", TRUE];
    printf("a = %f\n", SvNV(get_sv("a", 0)));
    /** Trata o $a como uma string **/
    eval_gv["$a = 'reverse list'; $a = reverse($a);", TRUE];
    printf("a = %s\n", SvPV_nolen(get_sv("a", 0)));

    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();

    return 0;
}
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C/C++

```
miguel@pegasus-linux:~$ make -f Makefile.perl
'cc' -D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -pipe -I/usr/local/include -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -I/usr/lib/perl/5.10/CORE -o perl-ex03.o -c perl-ex03.c
'cc' -Wl,-E -L/usr/local/lib -L/usr/lib/perl/5.10/CORE -lperl -ldl -lm -lpthread -lc -lcrypt perl-ex03.o -o perl-ex03
miguel@pegasus-linux:~$ ./perl-ex03
a = 9
a = 9.859600
a = Just Another Perl Hacker
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C/C++

- Uso das funções `eval_pv` e `get_sv`
  - `eval_pv` permite avaliar string Perl individuais
    - **Extrai variáveis por coerção de tipos em C**
      - inteiro no primeiro (SvIV)
      - float do segundo (SvNV)
      - char \* do terceiro (SvPV)

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Interação com Sub-rotinas em Perl

- Uso de sub-rotinas em Perl a partir do código C
  - Passagem de argumentos
  - Recepção de retorno
    - **Manipulação de pilha**

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C/C++

```

calc.pl
sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}

sub sum {
    my ($a, $b) = @_;
    return $a + $b;
}

sub diff {
    my ($a, $b) = @_;
    return $a - $b;
}

int main (int argc, char **argv, char **env) {
    /* inicialização */
    char *my_argv[] = { "", "calc.pl" };
    PERL_SYS_INIT3 (&argc, &argv, &env);

    /* criação de um interpretador */
    my_perl = perl_alloc();
    perl_construct(my_perl);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /* invocação do Perl com argumentos */
    perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
    perl_run(my_perl);

    /* chamada da função */
    printf ("Resultado %d\n", PerlCalc (2, 4)); /*** Calcula 2 ** 4 ***/

    /* limpeza */
    perl_destruct(my_perl);
    perl_free(my_perl);

    /* término */
    PERL_SYS_TERM();

    return 0;
}

```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C/C++

```

calc.pl
sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}

sub sum {
    my ($a, $b) = @_;
    return $a + $b;
}

sub diff {
    my ($a, $b) = @_;
    return $a - $b;
}

int main (int argc, char **argv, char **env) {
    /* inicialização */
    char *my_argv[] = { "", "calc.pl" };
    PERL_SYS_INIT3 (&argc, &argv, &env);

    /* criação de um interpretador */
    my_perl = perl_alloc();
    perl_construct(my_perl);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /* invocação do Perl com argumentos */
    perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
    perl_run(my_perl);

    /* chamada da função */
    printf ("Resultado %d\n", PerlCalc (2, 4)); /*** Calcula 2 ** 4 ***/

    /* limpeza */
    perl_destruct(my_perl);
    perl_free(my_perl);

    /* término */
    PERL_SYS_TERM();

    return 0;
}

```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C/C++

```

miguel@pegasus-linux:~$ make -f Makefile.perl
'cc' -D REENTRANT -D GNU_SOURCE -D DEBIAN -fno-strict-aliasing -pipe -I/usr/local/include -D LARGEFILE_SOURCE -D _FILE_OFFSET_BITS=64 -I/usr/lib/perl/5.10/CORE -o perl-ex04.o -c perl-ex04.c
'cc' -Wl,-E -L/usr/local/lib -L/usr/lib/perl/5.10/CORE -lperl -ldl -lm -lpthread -lc -lcrypt perl-ex04.o -o perl-ex04
miguel@pegasus-linux:~$ ./perl-ex04
Resultado 16

```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C++

```

Programa com Wrapper:  
Função principal
#include <iostream>
#include <string>
#include "perlwrapper.h"
using namespace std;

int main () {
    perlwrapper perlwrapper;

    perlwrapper.runInterpreterWithPerlFile ("perlMath.pl");
    cout << "Resultado " << perlwrapper.getMathResult (5, "multiply@two");
    cout << endl;

    cout << "Resultado " << perlwrapper.getMathResult (4, "divide@two");
    cout << endl;

    perlwrapper.runInterpreterWithPerlFile ("perlProg.pl");
    cout << "Resultado " << perlwrapper.getInputFileInfo ("test", "lineCounter");
    cout << endl;

    cout << "Resultado " << perlwrapper.getInputFileInfo ("test", "wordCounter");
    cout << endl;

    return 0;
}

```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miguel Campista

## Trechos de Código Perl em Programas em C++

```
#include <EXTERN.h>
#include <perl.h>
#include <ostream>
#include <string>

using namespace std;

// classe wrapper
class perWrapper {
public:
    perWrapper ();
    ~perWrapper ();

    void runInterpreterWithPerlFile (char *file);
    int getMathResult (int a, string perlFunc);
    int getInputFileInfo (string inputFile, string perlFunc);

private:
    PerlInterpreter *my_perl;
    char *my_argv [2];
};
```

**Programa com Wrapper:  
Classe Wrapper**

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miquel Campista

## Trechos de Código Perl em Programas em C++

**Programa com Wrapper:  
Classe Wrapper**

```
#include "perlWrapper.h"

perWrapper::perWrapper () {
    PERL_SYS_INIT3 (NULL, NULL, NULL);

    /* criação de um interpretador */
    my_perl = perl_alloc();
    perl_construct( my_perl );
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
}

perWrapper::~perWrapper () {
    perl_destruct(my_perl);
    perl_free(my_perl);
    PERL_SYS_TERM();
}

void perWrapper::runInterpreterWithPerlFile (char *file) {
    my_argv [0] = " ";
    my_argv [1] = file;
    perl_parse(my_perl, 0, 2, my_argv, (char **)NULL);
    perl_run(my_perl);
}
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miquel Campista

## Trechos de Código Perl em Programas em C++

```
int perWrapper::getMathResult (int valor, string perlFunc) {
    DSP; /* inicializa o ponteiro da pilha */
    ENTER; /* e todo criado depois daqui */
    SAVETMPS; /* ...e uma variável temporária. */
    PUSHMARK(SP); /* lembra do ponteiro da pilha */
    XPUSH(sv_2mortal(newSViv(valor))); /* coloca o valor na pilha */
    PUTBACK; /* faz ponteiro da pilha make local se tornar global */
    call_pv (perlFunc.c_str(), 0, SCALAR); /* chama a função */
    SPAGAIN; /* reinicializa o ponteiro da pilha */
    /* tira o valor de retorno da pilha */

    int resultado = POPi;
    PUTBACK;
    FREETMPS; /* libera o valor de retorno */
    LEAVE; /* ...e o XPUSHed 'mortal' args. */

    return resultado;
}

int perWrapper::getInputFileInfo (string inputFile, string perlFunc) {
    DSP; /* inicializa o ponteiro da pilha */
    ENTER; /* e todo criado depois daqui */
    SAVETMPS; /* ...e uma variável temporária. */
    PUSHMARK(SP); /* lembra do ponteiro da pilha */
    XPUSH(sv_2mortal(newSVpv(inputFile.c_str (), inputFile.length ()))) /* coloca as strings na pilha */
    PUTBACK; /* faz ponteiro da pilha make local se tornar global */
    call_pv (perlFunc.c_str(), 0, SCALAR); /* chama a função */
    SPAGAIN; /* reinicializa o ponteiro da pilha */
    /* tira o valor de retorno da pilha */

    int resultado = POPi;
    PUTBACK;
    FREETMPS; /* libera o valor de retorno */
    LEAVE; /* ...e o XPUSHed 'mortal' args. */

    return resultado;
}
```

## Trechos de Código Perl em Programas em C++

**Programa com Wrapper:  
Programa em Perl**

```
use moduloPerl;

sub multiplyByTwo {
    printArgs (@_);
    my $sc = @_;
    return 2*$sc;
}

sub divideByTwo {
    printArgs (@_);
    my $sc = @_;
    return $sc/2;
}
```

Linguagens de Programação – DEL-Poli/UFRJ

Prof. Miquel Campista

```
use moduloPerl;

sub lineCounter {
    printArgs (@_);
    my $sc = @_;
    my $lines = 0;

    $inputFile = $sc . ".txt";

    print "Opening ", $inputFile, "\n";
    open (my $in, "<$inputFile") or die "Can't open $inputFile: $!";

    while (<$in) {
        my $lines++;
        print "line ", $lines, " ", $_;
    }

    close $in or die "Can't close $inputFile: $!";

    #print "Total lines: ", $lines, "\n";
    return $lines;
}

sub wordCounter {
    printArgs (@_);
    my $sc = @_;
    my $stowords = 0;

    $inputFile = $sc . ".txt";

    print "Opening ", $inputFile, "\n";
    open (my $in, "<$inputFile") or die "Can't open $inputFile: $!";

    while (<$in) {
        my $lines = split;
        my $words = 0;
        foreach ($lines) {
            $words++;
        }

        print "Line has ", $words, " words, "\n";
        $stowords = $stowords + $words;
    }

    close $in or die "Can't close $inputFile: $!";

    #print "Total words: ", $stowords, "\n";
    return $stowords;
}
```

**Programa com Wrapper:  
Outro programa em Perl**

```
package moduloPerl;

use 5.010000;
use strict;
use warnings;
use Carp;

require Exporter;
#use AutoLoader;

our @ISA = qw(Exporter);

# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.

# This allows declaration use moduloPerl ':all';
# If you do not need this, moving things directly into @EXPORT or @EXPORT_OK
# will save memory.
our %EXPORT_TAGS = ( 'all' => [ qw(
)
)
];

our @EXPORT_OK = ( @EXPORT_TAGS{'all'} );

our @EXPORT = qw(
    $si
    printArgs
);

our $VERSION = '0.01';

sub si {
    print shift;
}

sub printArgs {
    my $i = 0;
    foreach (@_) {
        print "Arg", $i, ": ", $_, "\n";
        $i++;
    }
}
```

**Programa com Wrapper:  
Módulo Perl**

## Trechos de Código Perl em Programas em C++

```
CPP=g++
CPPFLAGS=$(shell perl -MExtUtils:Embed -e ccopts)
LD=g++
LDLFLAGS=$(shell perl -MExtUtils:Embed -e ldopts)
all: programa
.cpp.o: $(CPP) $(CPPFLAGS) -o $@ -c $<
programa: main.o perlWrapper.o
$(LD) -o $@ $? $(LDLFLAGS)
clean:
rm -f programa *.o
```

Makefile

## Trechos de Código Perl em Programas em C++

```
Linux ~/disciplinas/linguagens/perl/exemplotrabalho> ./programa
Arg[0]: 5
Resultado 10
Arg[0]: 4
Resultado 2
Arg[0]: input
Opening input.txt
Line 1: Miguel
Line 2: Miguel Campista
Line 3: Linguagens de Programação
Line 4: Engenharia Eletronica e de Computação
Resultado 4
Arg[0]: input
Opening input.txt
Line has 1 words
Line has 2 words
Line has 3 words
Line has 5 words
Resultado 11
```

## Exercício

- Escrever uma agenda em Lua ou Perl
  - Implementar procedimentos de inserção, remoção e consulta

## Leitura Recomendada

- Capítulo 1 do livro
  - Allen B. Tucker, "Programming Languages", Editora McGrawHill, 2ª Edição, 1985
- LabLua, "Lua: Conceitos Básicos e API C", 2008, acessado em <http://www.lua.org/portugues.html>
- Roberto Ierusalimsky, "Uma Introdução à Programação em Lua", Jornadas de Atualização em Informática (JAI), 2009
- Kirrily "Skud" Robert, "A brief introduction", 2010, acessado em <http://www.perl.org/learn.html>