

Linguagens de Programação

Prof. Miguel Elias Mitre Campista

`http://www.gta.ufrj.br/~miguel`

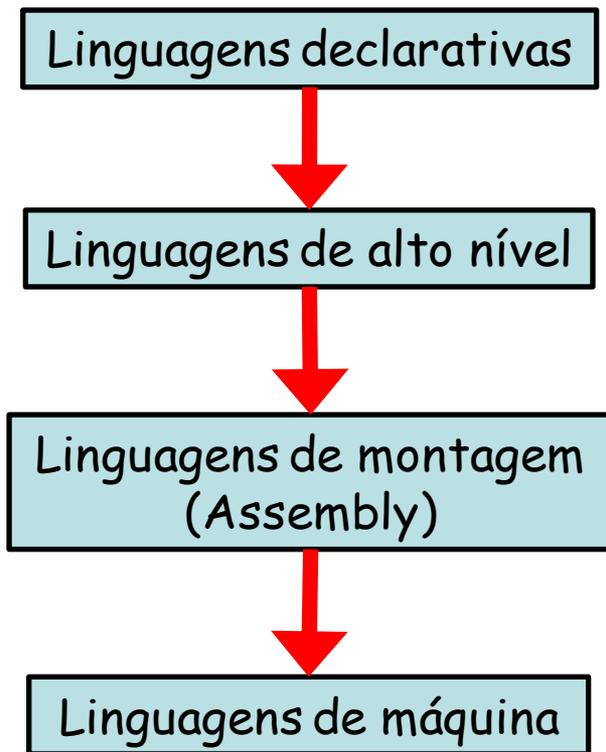
Parte II

Programação em Linguagens Estruturadas

Relembrando da Última Aula...

- **Algoritmo**
 - Um procedimento bem definido computacionalmente que recebe uma entrada e produz uma saída
- **Estrutura de dados**
 - São formas de armazenar e organizar dados para facilitar o acesso e possíveis modificações
- **Programa computacional**
 - É um algoritmo expresso em uma linguagem de programação

Níveis de Linguagens de Programação



Níveis de Linguagens de Programação

- Linguagens declarativas
 - Linguagens expressivas como a linguagem oral
 - Expressam o que fazer ao invés de como fazer
- Linguagens de alto nível
 - Linguagens típicas de programação
 - Permitem que algoritmos sejam expressos em um nível e estilo de escrita fácil para leitura e compreensão
 - Possuem características de portabilidade já que podem ser transferidas de uma máquina para outra
- Linguagens de montagem e linguagens de máquina
 - Linguagens que dependem da arquitetura da máquina
 - Linguagem de montagem é uma representação simbólica da linguagem de máquina associada

Níveis de Linguagens de Programação

Pascal	Linguagem de Montagem	Linguagem de Máquina
Z:= W+X*Y	LOAD 3,X	41 3 0C1A4
	MULTIPLY 2,Y	3A 2 0C1A8
	ADD 3,W	1A 3 0C1A0
	STORE 3,Z	50 3 0C1A4

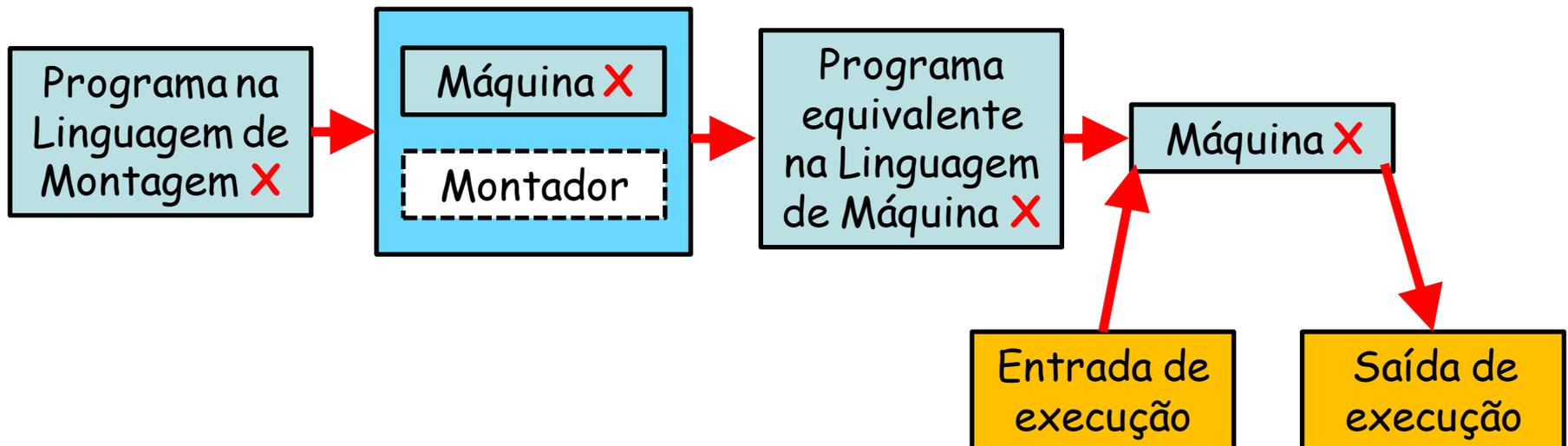
Níveis de Linguagens de Programação

Pascal	Linguagem de Montagem	Linguagem de Máquina
Z:= W+X*Y	LOAD 3,X	41 3 0C1A4
	MULTIPLY 2,Y	3A 2 0C1A8
	ADD 3,W	1A 3 0C1A0
	STORE 3,Z	50 3 0C1A4

Correspondência 1 para 1

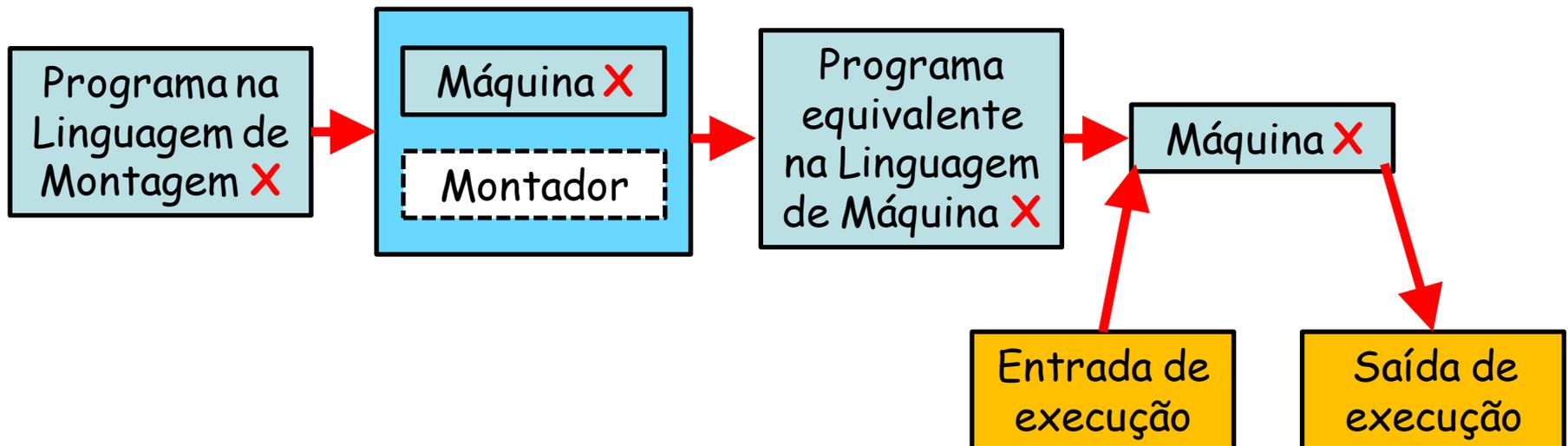
Programa Montador

- Responsável por converter o programa na linguagem de máquina correspondente



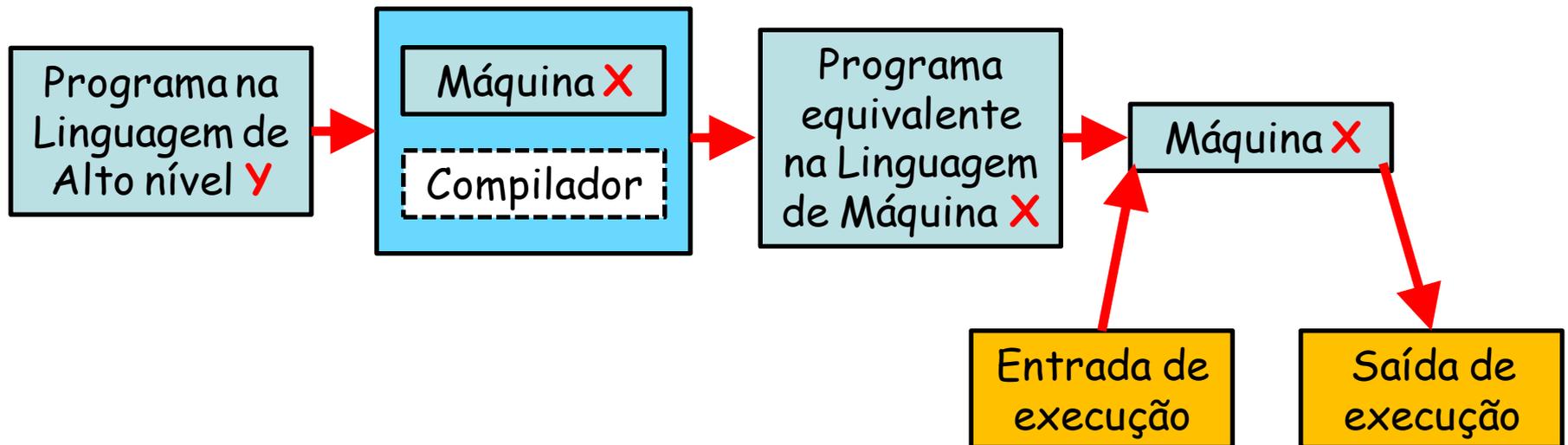
Programa Montador

- Responsável por converter o programa na linguagem de máquina correspondente

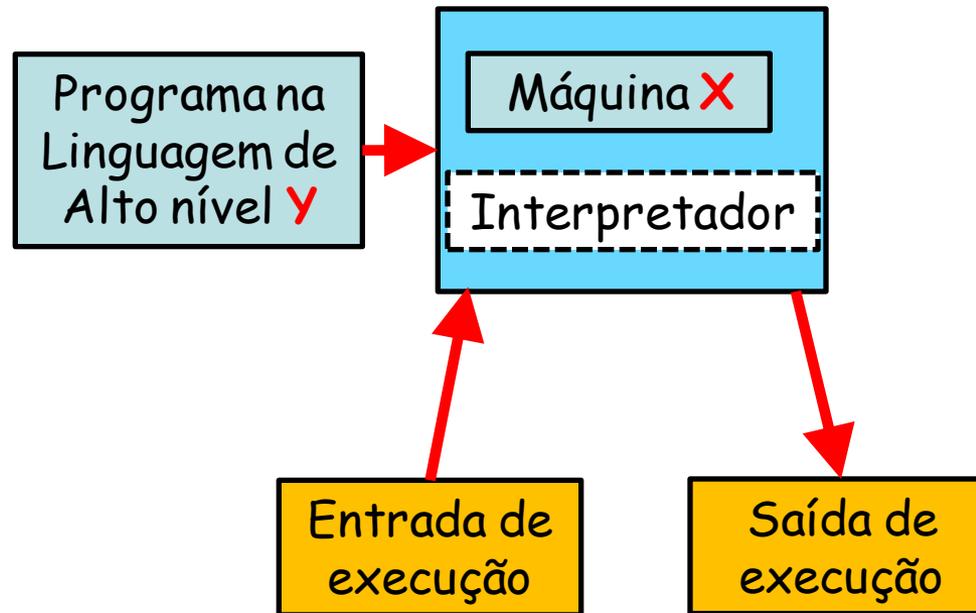


➔ **Como ficaria o programa compilador? E o interpretador?**

Programa Compilador



Programa Interpretador



Paradigmas de Programação em Alto Nível

Programação Imperativa

Programação Imperativa

- Chamada também de programação algorítmica
- Descreve a computação em detalhes em termos de **sentenças** que mudam o estado do programa
 - Define sequências de comandos para o computador executar
 - Semelhante a uma linguagem oral imperativa:
 - **Chefe:** - Some dois números!
 - **Chefe:** - Exiba o resultado!
 - **Chefe:** - Volte ao seu trabalho anterior!
 - **Chefe:** - etc.
 - **Relembrando:** Estado de um programa é definido pelas suas estruturas de dados e variáveis

Sentenças

- Menor elemento em uma linguagem de programação imperativa capaz de realizar mudança de estado
 - Sentença simples
 - **Atribuição:** $a = a + 1$
 - **Chamada:** `funcao()`
 - **Retorno:** `return 0`
 - **Desvio:** `goto 1`
 - **Asserção:** `assert(a == 0)`

Sentenças

- Menor elemento em uma linguagem de programação imperativa capaz de realizar mudança de estado

- Sentença composta

- **Bloco:** begin

```
write('Y');
```

```
end
```

- **Condição:** if $a > 3$ then

```
write('Y');
```

```
else
```

```
write('N');
```

```
end
```

Sentenças

- Menor elemento em uma linguagem de programação imperativa capaz de realizar mudança de estado

- Sentença composta

- **Chaveamento:** switch (c)

```
case 'a':  
    alert(); break;
```

```
case 'q':  
    quit(); break;
```

```
end
```

- **Laço de repetição:** while a>3 do

```
write('Y');
```

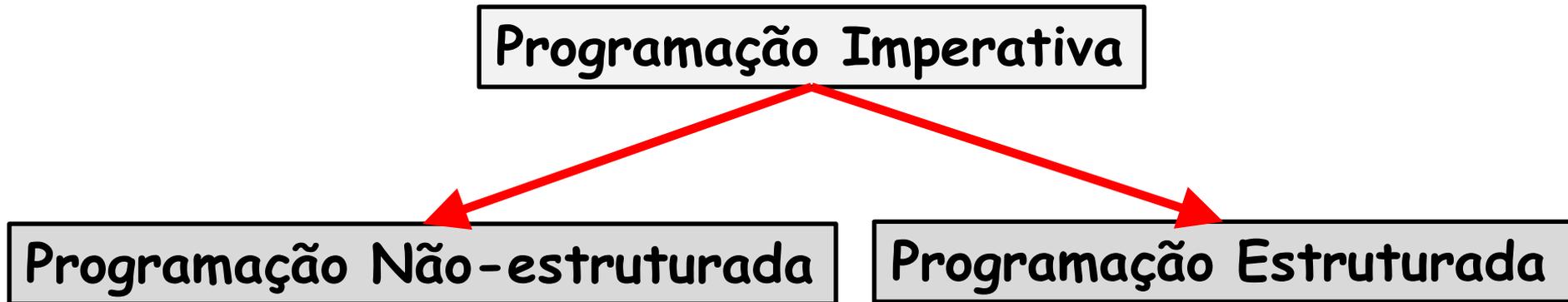
```
end
```

Sentenças

- Diferenças nas sintaxes
 - Separação de sentenças
 - Término de sentenças

Linguagem	Separação/Terminação de Sentença
Cobol	. (ponto)
C e C++	; (ponto e vírgula)
Java, Perl	; (ponto e vírgula)
Python	Nova linha
Lua	Espaço em branco (separando)

Paradigmas de Programação em Alto Nível



Programação Não-estruturada

- Tipo de programação **imperativa**
 - Código caracterizado pela presença de sentenças do tipo **goto**
 - Cada sentenças ou linha de código é identificada por um rótulo ou um número
 - **Chefe:** 10 - Imprimir resultado
 - **Chefe:** 20 - Se $A+B$ for maior que C
 - **Chefe:** 30 - Vá para 10
 - **Chefe:** 40 - Se $A+B$ for menor que C
 - **Chefe:** 50 - Some mais 1
 - Oferece liberdade de programação
 - Entretanto...
 - Torna o código complexo

Programação Não-estruturada

- Tipo de programação **imperativa**
 - Código caracterizado pela presença de sentenças do tipo **goto**
 - Cada sentenças ou linha de código é identificada por um rótulo ou um número
 - **Chefe:** 10 - Imprimir resultado
 - **Chefe:** 20 - Se $A+B$ for maior que C
 - **Chefe:** 30 - Vá para 10
 - **Chefe:** 40 - Se $A+B$ for menor que C
 - **Chefe:** 50 - Some mais 1
 - Oferece liberdade de programação
 - Entretanto...
 - Torna o código complexo

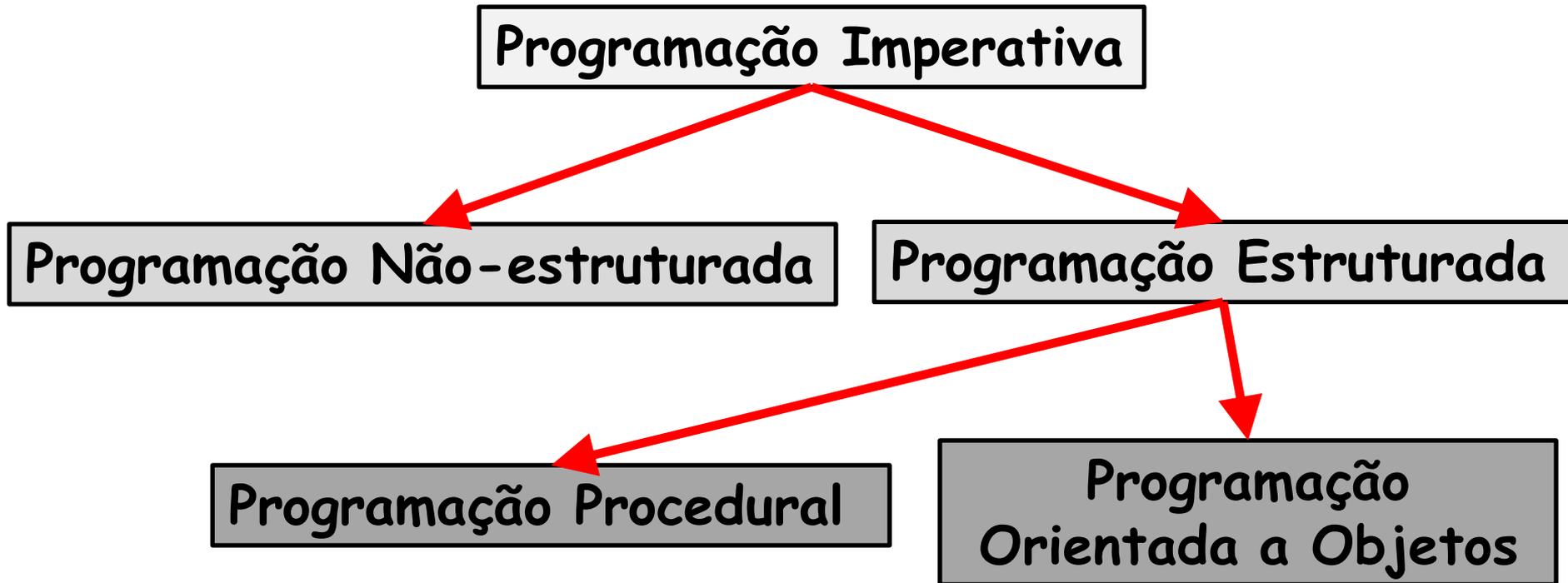


O "goto" e o "if", por algum tempo, eram as únicas estruturas de controle das linguagens de programação. Não existia, p.ex., o "while" nem o "se-então-senão"!

Programação Estruturada

- Tipo de programação **imperativa**
 - Basicamente não utiliza sentenças do tipo **goto**
 - **Dispensa os rótulos**
 - **Chefe:** - Se $A+B$ for maior que C
 - **Chefe:** - Imprimir resultado
 - **Chefe:** - Caso contrário
 - **Chefe:** - Some mais 1

Paradigmas de Programação em Alto Nível



Programação Procedural

- Tipo de programação **imperativa e estruturada** baseada em **procedimentos**
 - Procedimentos são sinônimos de funções, métodos ou sub-rotinas
 - Ex.: Linguagem C
 - **Chefe:** - somar(a, b)
 - **Chefe:** - imprimir("terminado!")
 - **Chefe:** - voltar

Programação Procedural

- Tipo de programação **imperativa e estruturada** baseada em **procedimentos**
 - Procedimentos são sinônimos de funções, métodos ou sub-rotinas
 - Ex.: Linguagem C
 - **Chefe:** - somar(a, b)
 - **Chefe:** - imprimir("terminado!")
 - **Chefe:** - voltar



Resolve o problema por partes, subdividindo-o até que a subdivisão seja simples o suficiente para ser resolvida por apenas um procedimento.

Programação Procedural

- Uso de procedimentos permite:
 - Reuso de procedimentos em diferentes partes do código
 - **Chefe:** `r = somar(a,b)`
 - **Chefe:** `imprimir(r)`
 - **Chefe:** `r = somar(a,r)`

Programação Procedural

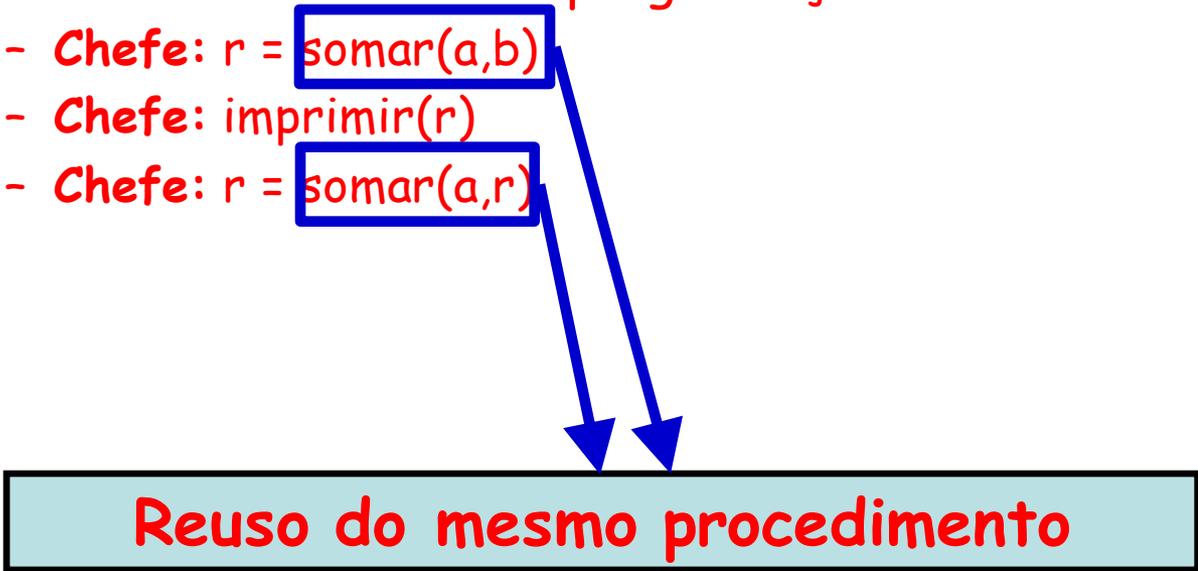
- Uso de procedimentos permite:
 - Reuso de procedimentos em diferentes partes do código!

- Aumenta a eficiência da programação

- **Chefe:** `r = somar(a,b)`

- **Chefe:** `imprimir(r)`

- **Chefe:** `r = somar(a,r)`



Reuso do mesmo procedimento

Programação Orientada a Objetos

- Tipo de programação **imperativa e estruturada**, porém...
 - Enquanto a programação procedural é estruturada em...
 - **Procedimentos**
 - Estruturas de dados e algoritmos
 - A programação orientada a objetos é estruturada em...
 - **Classes e objetos**
 - Objetos encapsulam estruturas de dados e procedimentos

Programação Orientada a Objetos

- Tipo de programação **imperativa e estruturada**, porém...
 - Enquanto a programação procedural é estruturada em...
 - **Procedimentos**
 - Estruturas de dados e algoritmos
 - A programação orientada a objetos é estruturada em...
 - **Classes e objetos**
 - Objetos encapsulam estruturas de dados e procedimentos

Procedural

```
main() {  
    define_carro();  
    entra_carro();  
    sair_carro();  
}
```

Orientada a objetos

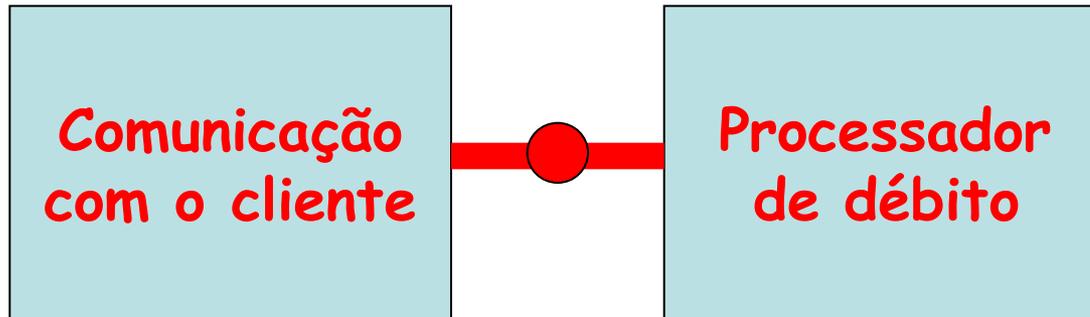
```
main() {  
    Carro carro;  
    carro.entrar();  
    carro.sair();  
}
```

Outros Paradigmas de Programação...

- Programação baseada em eventos
 - Fluxo do programa é determinado pelo surgimento de eventos
 - Eventos podem ser disparados pela recepção de mensagens ou expiração de temporizadores
- Programação orientada a agentes
 - Programa é estruturado em agentes
 - Agente é uma abstração de um software capaz de tomar decisões autônomas
 - Ao invés de métodos e atributos, um agente possui **comportamento**

Outros Paradigmas de Programação...

- Programação orientada a componentes
 - Programa cujo o objetivo é unir blocos funcionais
 - Diferente da orientação a objetos, não há a preocupação em modelar objetos como objetos da vida real



Linguagens de Programação Estruturadas

- Assim como os programadores e as aplicações...
 - As linguagens são especializadas dependendo da aplicação para qual foram desenvolvidas
 - **Aplicações científicas**
 - Especializadas em manipulação de números e vetores
 - Empregam ferramentas matemáticas e estatísticas
 - Requerem mais processamento que entrada e saída de dados
 - » Ex.: Pascal, Fortran, APL
 - **Aplicações de processamento de dados**
 - Especializadas na criação, manutenção, mineração e resumo de dados em registros ou em arquivos
 - Requerem entrada e saída e nem tanto de processamento
 - » Exs.: Cobol e PL/I

Linguagens de Programação Estruturadas

- Assim como os programadores e as aplicações...
 - As linguagens são especializadas dependendo da aplicação para qual foram desenvolvidas
 - **Aplicações de processamento de texto**
 - Especializadas em manipulação de textos em linguagem natural, ao invés de números e dados
 - » Ex.: SNOBOL
 - **Aplicações de inteligência artificial**
 - Especializadas na emulação de comportamento inteligente
 - Incluem algoritmos de jogos, reconhecimento de padrão etc.
 - » Exs.: LISP e Prolog

Linguagens de Programação Estruturadas

- Assim como os programadores e as aplicações...
 - As linguagens são especializadas dependendo da aplicação para qual foram desenvolvidas
 - **Aplicações de programação de sistemas**
 - Especializadas no desenvolvimento de programas para interface entre o programa e o hardware da máquina
 - Lidam com eventos imprevistos como erros
 - Incluem compiladores, interpretadores, montadores etc.
 - » Exs. Ada e Modula-2

Linguagens de Programação Estruturadas

- Assim como os programadores e as aplicações...
 - As linguagens são especializadas dependendo da aplicação para qual foram desenvolvidas
 - **Aplicações de programação de sistemas**
 - Especializadas no desenvolvimento de programas para interface entre o programa e o hardware da máquina
 - Lidam com eventos imprevistos como erros
 - Incluem compiladores, interpretadores, montadores etc.
 - » Exs. Ada e Modula-2



Apesar da motivação inicial de desenvolvimento, com o passar do tempo, as linguagens se tornaram mais versáteis e completas. Ex.: C++, Lua e Python

Critérios de Avaliação e Comparação de Linguagens

- Expressividade
 - Capacidade de refletir com clareza o seu objetivo
 - Ex.: $C = A + B$
 $C := A + B$
(SETQ C(+ A B))
ADD A, B GIVING C
- Delineamento
 - Capacidade da linguagem não apresentar ambiguidades
- Estruturas e tipos de dados
 - Suporte a diferentes estruturas de dados e tipos
- Modularidade
 - Suporte à subprogramação e à extensão

Critérios de Avaliação e Comparação de Linguagens

- Entrada e saída
 - Suporte a diferentes maneiras de acesso a dados e arquivos
- Portabilidade
 - Dependência de máquinas específicas
- Eficiência
 - Velocidade de compilação/tradução e execução
- Generalidade
 - Capacidade de uso em diferentes aplicações
- Simplicidade de aprendizado

Primeiras Linguagens

- Lua
- Perl

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

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
```

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

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!"*

Linguagem Lua

- Trecho
 - Peçaço 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>

Primeiro Exemplo em Lua

- Programa: HelloWorld.lua

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

- Compilação+Execução: lua HelloWorld.lua

```
shell>$ lua HelloWorld.lua
```

```
Hello, world!
```

```
shell>$
```

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>$
```

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/>

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!  
>
```

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`

Variáveis em Lua

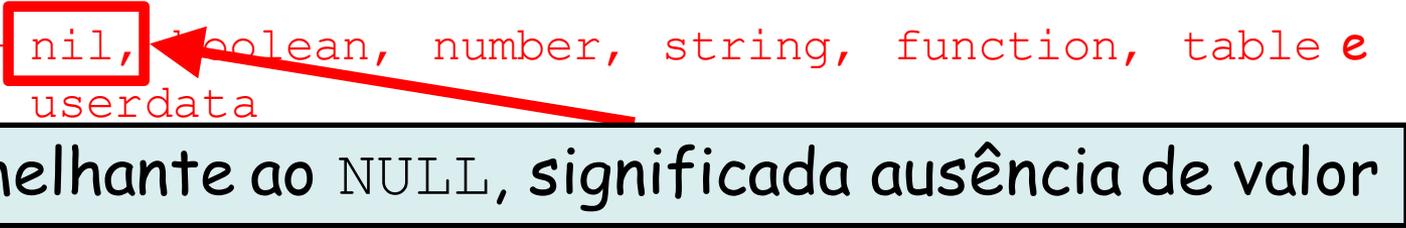
- Escopo

- 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

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

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)

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]]`

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

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.)

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

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

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

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"
```

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

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
```

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}`

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>$
```

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
```

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

Tabelas em Lua

- Uso de função em tabelas

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")
```

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

Tabelas em Lua

- Uso de função em tabelas

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")

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

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

```
shell>$ lua funcaoTable.lua
```

```
1 2 function: 0x1cb2bb0
```

```
Hello, Miguel
```

```
1 1 function: 0x1cb2bb0 function: 0x1cb2bb0
```

```
Hello, Elias
```

```
Hello, Campista
```

```
shell>$
```

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]))
```

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>$
```

Tabelas em Lua

- Inserção em tabelas associativas

Arquivo: assocTable.lua

```
--> Declaração e inicialização da tabela t
local t = {x=1, y=2, z=3}
print (t.x, t.y, t.z)

function t.hello (nome)
    print ("Hello,", nome)
end

t.hello ("Miguel")
```

Tabelas em Lua

- Inserção em tabelas associativas

Arquivo: assocTable.lua

```
--> Declaração e inicialização da tabela t
local t = {x=1, y=2, z=3}
print (t.x, t.y, t.z)

function t.hello (nome)
    print ("Hello,", nome)
end

t.hello ("Miguel")
```

Inserção a posteriori
em uma tabela
associativa

```
shell>$ lua assocTable.lua
1      2      3
Hello, Miguel
shell>$
```

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
```

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

Programa iteradores.lua

```
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
```

```
shell>$ lua iteradores.lua
```

```
1
3
5
x      1
y      3
z      5
```

```
aluno
eletrônica
linguagens
```

```
shell>$
```

Estruturas de Controle em Lua

- `ipairs` vs `pairs`

Programa `pairsvsipairs.lua`

```
a1 = {[1]=2, [2]=3, [4]=7, [5]=9, z=5}
for i, v in ipairs (a1) do
    print (i, v)
end
print ("\n")
for k, v in pairs (a1) do
    print (k, v)
end
```

E agora, o que é
impresso na tela?

Estruturas de Controle em Lua

- `ipairs` vs `pairs`

Programa `pairsvsipairs.lua`

```
a1 = {[1]=2, [2]=3, [4]=7, [5]=9, z=5}
```

```
for i, v in ipairs (a1) do  
    print (i, v)  
end
```

```
print ("\n")
```

```
for k, v in pairs (a1) do  
    print (k, v)  
end
```

```
shell>$ lua pairsvsipairs.lua
```

```
1      2
```

```
2      3
```

```
5      9
```

```
1      2
```

```
2      3
```

```
4      7
```

```
z      5
```

```
shell>$
```

`ipairs` só exhibe pares com chaves numerais em ordem começando do [1]

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
```

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

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

Uso de Funções como Argumento

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

```
shell>$ lua argFuncao.lua
```

```
2
```

```
3
```

```
4
```

```
shell>$
```

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

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>$
```

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
```

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

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)`

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

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

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

```
2
2
```

Passagem de parâmetro por valor

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

```
ERRO! a = nil...
```

```
shell>$
```

Variável criada localmente após a declaração da função

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>$
```

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`)

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

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, 2*(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
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-codigo>`

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

- `array`

- Representa uma lista

- `hash`

- Representa um conjunto

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`

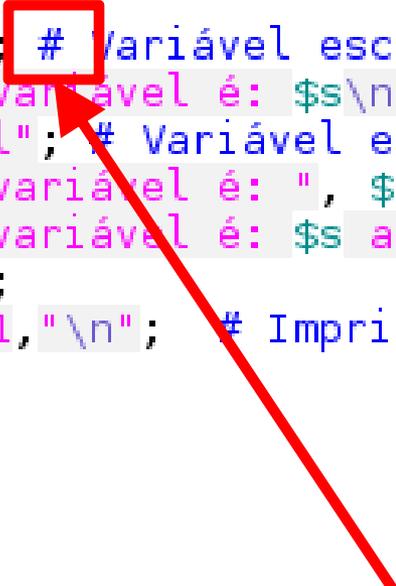
```
my $s = 3; # Variável escalar receber um inteiro
print "A variável é: $s\n"; # Imprime 3
$s = "perl"; # Variável escalar receber uma string
print "A variável é: ", $s, " agora\n"; # Imprime perl
print "A variável é: $s agora\n";
$s = 1.23;
print $s+1, "\n"; # Imprime 2.23
```

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`

```
$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>$
```

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 [1..$#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", "llama", "owl"
print @animals [1..$#animals], "\n"; # Imprime "llama" e "owl"
print @animals; # Imprime tudo
```

- Interpretação:
`perl vetores.pl`

```
shell>$ perl vetores.pl
camel
llama
1.23
camellama
camellamaowl
llamaowl
camellamaowl
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  
vemelha  
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

```
# Variavel escalar que recebe uma referencia
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

```
# Variavel escalar que recebe uma referencia
my $variables = {
    scalar => {
        description => "single item",
        sigil => '$',
```

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

```
hash => {
    description => "key/value pairs",
    sigil => '%',
},
};

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

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

Exemplo Usando Referências

- Programa: `refs.pl`

```
# Isso
$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";
```

Exemplo Usando Referências

- Programa: `refs.pl`

```
# Isso
$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 5
shell>$
```

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

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

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

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

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

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

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

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

```
shell>$ perl refs2.pl
```

```
b
b
```

```
exemplo
```

```
2
2
5
5
```

```
exemplo
```

```
2
2
5
5
```

```
8
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";
```

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>$
```

Operadores em Perl

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

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) {...}
```

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;  
}  
foreach (@vetor) { #Varre o vetor  
    print "Elemento $_"; # $_ var. padrão  
}  
foreach (keys %hash) {  
    print "Chaves $_";  
}  
foreach my $k (keys %hash) {  
    print "Chaves $k"; # Sem a var. padrão  
}
```

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;
```

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;
```

Obriga uso das chaves



Dispensa uso das chaves



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);
```

Sétimo Exemplo em Perl

```
sub ordena {  
    ($v1, $v2, $v3) = sort @_;  
    return ($v1, $v2, $v3);  
}  
  
my @v = (3, 2, 1);  
  
print ordena (@v), "\n";
```

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>$
```

Passagem de Parâmetro no Perl

- Pode ser por valor ou referência
 - Usando Escalares

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

```
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      EscalaRef = 4
```

```
shell>$
```

Passagem de Parâmetro no Perl

- Pode ser por valor ou referência
 - Usando Arrays

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

```
sub retornaArrayRef {  
    my $r = $_[0]; # Referência é escalar  
    print $r, "\t", @{$r}, "\t", $_[0]->[0], "\t", @{$_[0]}[1], "\n";  
    @{$r}[0]++; $r->[1]++;  
    print $r, "\t", @{$r}, "\t", $_[0]->[0], "\t", @{$_[0]}[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>$
```

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(0x1870ea0)  y5x4      4          5
HASH(0x1870ea0)  y6x5      5          6
Ref = y6x5 HashRef = y6x5
shell>$
```

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

sub retornaHashRef {
    my $r = $_[0]; # Referência é escalar
    print $r, "\t", %{$r}, "\t", $_[0]->{"x"}, "\t", %{$r}{y}, "\n";
    %{$r}{x}++; $r->{y}++;
    print $r, "\t", %{$r}, "\t", $_[0]->{"x"}, "\t", %{$r}{y}, "\n";
    return $r; # Poderia retornar também $_[0]
}

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

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

Passagem de Parâmetro no Perl

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

```
use warnings;
use strict;

sub imprimeArgs {
    print $_[0], "\t", $_[1], "\t", $_[2], "\t", $_[3],
           "\t", $_[4], "\t", $_[5], "\t", $_[6], "\n";
    # Perl concatena todas as estruturas em um único array...
}

sub imprimePorRefs {
    my $sref = $_[0]; my $aref = $_[1]; my $href = $_[2];
    # Referência é escalar
    print @{$sref}, "\t", @{$aref}, "\t", @{$href}, "\n";
}

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

```
shell>$ perl funcoesConcatenadas.pl
```

```
1          2          3          y          5          x          4
1          23         y5x4
shell>$
```

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>$
```

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: $!";
```

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

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>$
```

Oitavo Exemplo em Perl

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

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

Operador <> lê apenas uma linha em contexto de escalar

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

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

Operador <> lê o arquivo todo em contexto de array

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

```
print "*** Arquiv. todo ***\n";
my @lines = <$in>;
print @lines,
```

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

>\$ perl arquivoES.pl

pl

Entrada

Saída

Usando

Arquivos

**

Exemplo

Entrada

Saída

Usando

Arquivos

shell>\$

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>$
```

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>$
```

Módulos

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

```
# Apresenta warnings durante execução caso haja pontenciais problemas
use warnings;
# Pára a execução caso haja pontenciais 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>$
```

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
```

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.

Módulos

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

```
use warnings;  
use strict;  
  
$DEBUG = 1;  
  
print $DEBUG, "\n";
```

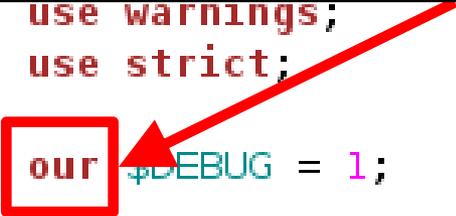
```
miguel@pegasus-linux:~$ perl global-strict.pl  
Global symbol "$DEBUG" requires explicit package name at global-strict.pl line 4  
.  
Global symbol "$DEBUG" requires explicit package name at global-strict.pl line 6  
.  
Execution of global-strict.pl aborted due to compilation errors.
```

Módulos

- A linguagem
- Carre

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 enésimo número da série de Fibonacci.
 - O enésimo 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; $i++) {
        print "Entre com o nome #$i:\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 NÚMERO 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 "Vou buscar \"gu\"\n";
if ($s =~ /gu/) {
    print "Achei gu\n";
    $s =~ s/gu/GU/;
    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
Vou buscar "gu"
Achei gu
Mudei para miGUel

Nome: miGUel

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

Primeiro nome: miGUel
Segundo nome: elias
```

Expressões Regulares

`\s` é caractere diferente de espaço em branco e `\S` caractere de espaço em branco

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

print "\n";

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

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

Nome: miGUel

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

Primeiro nome: miGUel
Segundo nome: elias
```

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

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/t/testeModule.t
Writing testeModule/Changes
Writing testeModule/MANIFEST
shell>$
```

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

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.

1;
__END__
# Below is stub documentation for your module. You'd better edit it!

=head1 NAME
```

interface

Criação da função oi

função

Nono Exemplo em Perl

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

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

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.

1;
__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

Nono Exemplo em Perl

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

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

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
```

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
```

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

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

VERSION=01

all: perl-ex$(VERSION)

perl-ex$(VERSION).o: perl-ex$(VERSION).c
    $(CC) $(CCFLAGS) -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)
```

```
VERSION=01
```

```
all: perl
```

```
perl-ex$
```

```
perl-ex$
```

```
$(LD) -o $@ $? $(LDFLAGS)
```

```
clean:
```

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

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

CC=g++

LD=g++

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);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /* invocação do perl com argumentos */
    int perl_argc = 3;
    char *code = "print scalar (localtime).\\\"\\n\\\"";
    char *perl_argv [] = {argv[0], "-e", code};
    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 -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-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

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 *args[] = {NULL};
    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, 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";
}
```

Chamada de Sub-rotinas Individuais

```
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-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
```

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**

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[] = { "", "-e", "0" };

    PERL_SYS_INIT3(&argc, &argv, &env);
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, 3, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_run(my_perl);

    /** Trata o $a como um inteiro **/
    eval_pv("$a = 3; $a **= 2", TRUE);
    printf("a = %d\n", SvIV(get_sv("a", 0)));
    /** Trata o $a como um float **/
    eval_pv("$a = 3.14; $a **= 2", TRUE);
    printf("a = %f\n", SvNV(get_sv("a", 0)));
    /** Trata o $a como uma string **/
    eval_pv("$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($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;
}
```

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
```

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`)

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

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

calc.pl

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

PerlInterpreter *my_perl;

int PerlCalc (int a, int b) {
    dSP;                                     /* inicializa o ponteiro da pilha */
    ENTER;                                   /* estudo criado depois daqui */
    SAVETMPS;                                /* ...é uma variável temporária. */
    PUSHMARK(SP);                            /* lembra do ponteiro de pilha */
    XPUSHs(sv_2mortal(newSViv(a)));          /* coloca a base na pilha */
    XPUSHs(sv_2mortal(newSViv(b)));          /* coloca o expoente na pilha */
    PUTBACK;                                  /* faz ponteiro da pilha make local se tornar global */
    call_pv("expo", G_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;
}
```

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

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

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

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

calc.pl

```
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;
}
```

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

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

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

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-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
```

Trechos de Código Perl em Programas em C++

```
#include <iostream>
#include <string>

#include "perlWrapper.h"

using namespace std;

int main () {
    perlWrapper perlwrapper;

    perlwrapper.runInterpreterWithPerlFile ("perlMath.pl");

    cout << "Resultado " << perlwrapper.getMathResult (5, "multiplyByTwo");
    cout << endl;
    cout << endl;

    cout << "Resultado " << perlwrapper.getMathResult (4, "divideByTwo");
    cout << endl;
    cout << endl;

    perlwrapper.runInterpreterWithPerlFile ("perlProg.pl");

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

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

    return 0;
}
```

**Programa com Wrapper:
Função principal**

Trechos de Código Perl em Programas em C++

Programa com Wrapper: Classe Wrapper

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

using namespace std;

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

    void runInterpreterWithPerlFile (char *file);

    int getMathResult (int a, string perlFunc);
    int getInputFileInfo (string inputFile, string perlFunc);

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

Trechos de Código Perl em Programas em C++

Programa com Wrapper: Classe Wrapper

```
#include "perlWrapper.h"

perlWrapper::perlWrapper () {
    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;
}

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

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

Trechos de Código Perl em Programas em C++

```
int perlWrapper::getMathResult (int valor, string perlFunc) {
    dSP;
    ENTER;
    SAVETMPS;
    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSViv(valor)));
    PUTBACK;
    call_pv (perlFunc.c_str(), G_SCALAR);
    SPAGAIN;

    int resultado = POPi;
    PUTBACK;
    FREETMPS;
    LEAVE;

    return resultado;
}

int perlWrapper::getInputFileInfo (string inputFile, string perlFunc) {
    dSP;
    ENTER;
    SAVETMPS;
    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSVpv(inputFile.c_str (), inputFile.length ()))));
    PUTBACK;
    call_pv (perlFunc.c_str(), G_SCALAR);
    SPAGAIN;

    int resultado = POPi;
    PUTBACK;
    FREETMPS;
    LEAVE;

    return resultado;
}
```

Trechos de Código Perl em Programas em C++

Programa com Wrapper:
Programa em Perl

```
use moduloPerl;

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

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

```
use moduloPerl;
```

```
sub lineCounter {
    printArgs (@_);

    my ($c) = @_;
    my $lines_ = 0;

    $inputFile = $c . ".txt";

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

    while (<$in>) {
        $lines_++;
        print "line ", $lines_, ": ", $_;
    }

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

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

Programa com Wrapper: Outro programa em Perl

```
sub wordCounter {
    printArgs (@_);

    my ($c) = @_;
    my $totWords_ = 0;

    $inputFile = $c . ".txt";

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

    while (<$in>) {
        my @line_ = split;
        my $words_ = 0;
        foreach (@line_) {
            $words_++;
        }
        print "Line has ", $words_, " words", "\n";
        $totWords_ = $totWords_ + $words_;
    }

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

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

Programa com Wrapper: Módulo Perl

```
package moduloPerl;

use 5.010001;
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(
    oi printArgs
);

our $VERSION = '0.01';

sub oi {
    print shift;
}

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

}
```

Trechos de Código Perl em Programas em C++

```
CPP=g++
CPPFLAGS=$(shell perl -MExtUtils::Embed -e ccopts)
LD=g++
LDFLAGS=$(shell perl -MExtUtils::Embed -e ldopts)

all: programa

.cpp.o:
    $(CPP) $(CPPFLAGS) -o $@ -c $<

programa: main.o perlWrapper.o
    $(LD) -o $@ $? $(LDFLAGS)

clean:
    rm -f programa *.o
```

Makefile

Trechos de Código Perl em Programas em C++

```
itaqua:~/disciplinas/linguagens/perl/exemplotrabalho3> ./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 Eletrônica 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 Ierusalimschy, "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>