PERL

LINGUAGENS DE PROGRAMAÇÃO

Janaina Dantas, Leonardo Ferreira e Raissa Endring

Sumário

1. Histórico
2. Características
3. Compilação/Interpretação
   3.1. Procedimento;
1. Sintaxe
   1. Variáveis;
   2. Escopo;
   3. Operadores;
   4. Controle de Fluxo;
   5. Estruturas de Repetição
5. Referências
6. Expressões Regulares
7. Sub-Rotinas
8. Arquivos
9. Coletor de Lixo
10. Módulos
11. Polimorfismo
12. Programação Orientada a Objetos
13. Tratamento de Exceções
14. Avaliação da Linguagem

Histórico

- Perl (Practical Extraction and Report Language).
- Criada por Larry Wall em 1987.

Utilizada para uma grande variedade de tarefas:

- *Processamento de Texto*
- *Desenvolvimento Web*
- *Administração de Sistemas*
- *Acesso a Banco de Dados.*

Histórico

- Processamento de texto em sistemas baseados em Unix era feito com diversas ferramentas (AWK, ‘sed’, C, e linguagens shell script).
- Wall juntou as vantagens de todas essas linguagens.
- Versão 5 lançada em 1994, à partir de então foi considerada por muitos uma linguagem “completa”.

Características

- Linguagem Simples.
- Portátil.
- Excelente para manipulação de textos.
- Suporte à programação procedural e Orientada a Objetos.
- Desenvolvimento rápido e eficiente.
- Multi-paradigma.
- Case Sensitive

Compilação/Interpretação

- Compilador que “pensa ser” interpretador.
- Existe um estágio de otimização em que o código de programa é compilado e transformado em código executável para que a linguagem seja realmente eficiente. Entretanto, ela não grava esse código em um arquivo executável separado. Ao invés disso o código executável apenas é copiado pra memória, sendo depois utilizado.

Compilação/Interpretação

- Combina desenvolvimento rápido de linguagem interpretada com execução eficiente de código compilado. Portanto, também agrega as desvantagens correspondentes, como a necessidade de compilar o programa toda vez que ele é executado.
- Pode ser tratada como linguagem interpretada.

Compilação/Interpretação - Procedimento

```
1 #!/bin/perl5.14.2
2
3 printf "Ola, professor!\n";
```

```
2012101378@labgrad02:~/Área de Trabalho$ perl prog.pl
Olá, professor!
2012101378@labgrad02:~/Área de Trabalho$
```

Sintaxe

- A primeira linha de um código em Perl deve começar com:

```
#!/usr/bin/perl
```

- Se declararmos uma variável mas não for atribuído nada, ela recebe um valor undef.
- Para comentar:

```
# Isto é um comentário em Perl
```

```
6 =for comment
7 print "Esse é um comentário de bloco!"
8 =cut
9
```

- Comandos são finalizados com ponto e vírgula (;)

```
3 print "Olá, professor!\n";
```

Sintaxe - Variáveis

- Basicamente existem 3 tipos:
  Escalar; Array; Hash;

- Sempre precedidos com um selo.
  $ para escalar.
  @ para array.
  % para hash.

Sintaxe - Variáveis

1. Escalares
- Representam um único valor;
- Podem ser string, inteiro, ou ponto flutuante.

```
3 $animal = "camelo"; $resposta = 42;
4 print "$animal\n";
5 print "O animal eh $animal\n";
6 print "O quadrado de $resposta eh ", $resposta * $resposta, "\n";
```

Sintaxe - Variáveis

2. Arrays

- Representam listas de valores.
- Índice inicial de uma váriavel array é zero.
- A variável especial <$#array> informa o índice do último elemento de um array.
- Funções Úteis:

    Push: Adiciona um elemento no fim da lista.
    Pop: Retira um elemento da última posição.
    Unshift: Adiciona um elemento no inicio.
    Shift: Remove do ínicio.

Sintaxe - Variáveis

```
3 @animais = ("camelo", "lhama", "coruja");
4 @numeros = (23, 42, 69);
5 @misturados = ("camelo", 42, 1.23);
6 print "$animais[0]\n"; # exibe "camelo"
7 print "$animais[1]\n"; # exibe "lhama"
8 print "$misturados[$#misturados]\n"; # último elemento, exibe 1.23
9
0 @animais[0,1]; # devolverá ("camelo", "lhama");
1 @animais[0..2]; # devolverá ("camelo", "lhama", "coruja");
2 @animais[1..$#animais]; # devolverá todos exceto o primeiro elemento
```

Sintaxe - Variáveis

3. Hashes

- Um hash representa uma coleção de pares de chave/valor.
- Você pode utilizar um espaço em branco e o operador “ => “ para uma leitura mais agradável.

```
3 %cores_de_frutas = ("morango", "vermelho", "banana", "amarelo");
4 %cores_de_frutas = ( morango => "vermelho", banana => "amarelo", );
5 $cores_de_frutas{"morango"}; # retorna "vermelho"
6
```

Sintaxe - Variáveis

- Você pode obter a lista de chaves ou lista de valores com as funções <keys()> e <values()>.
- As variáveis hash não tem uma ordem interna particular, porém você pode ordenar as chaves utilizando a função <sort()> e iterar entre elas.

```
my %cores_de_frutas = ( morango => "vermelho", banana => "amarelo" );
my @frutas = keys %cores_de_frutas;
my @cores = values %cores_de_frutas;
print "@frutas[0..1]\n"; #imprime banana morango
print "@cores[0..1]"; #imprime amarelo vermelho
```

Sintaxe - Variáveis

Existem algumas variáveis especiais, que são utilizadas para vários propósitos. As mais utilizadas são:

- $_ : variável padrão.
- @_: argumentos passados para subrotina.
- @ARGV: argumentos passados na linha de comando.
- %ENV: contém variáveis de ambiente.

Sintaxe - Variáveis

- A variável especial $_ é considerada por default em muitos operadores Perl, e tende a ser usada com muita freqüência.

```
@vetor = (100, 200, 300);
foreach (@vetor) {
  print $_;
}
```

- No exemplo acima, cada iteração da repetição (que percorrerá todos os elementos de vetor) colocará o seu corrente dentro de $_, que será impresso em seguida.

Sintaxe - Escopo

- Todos os exemplos utilizados até agora, utilizaram a sintaxe:

  $var = "valor";

  Porém, ao utilizar essa sintaxe, criamos apenas variáveis globais.

  Para criar variáveis dentro de um escopo léxico, utiliza-se:

```
1 my $a = "foo";
2 if ($alguma_condicao){
3         my $b = "bar";
4         print $a; # exibe "foo"
5         print $b; # exibe "bar"
6         }
7         print $a; # exibe "foo"
8         print $b; # nao exibe nada;
9         $b saiu de escopo.
```

Sintaxe - Operadores

Operador	Nome	Explicação	Exemplo
+	Adição	Soma o valor de 2 variáveis	$a=2; $b=3; $c=$a+$b; #$c=5
-	Subtração	Subtrai o valor de 2 variáveis	$a=2; $b=3; $c=$a-$b; #$c=-1
*	Multiplicação	Multiplica o valor de 2 variáveis	$a=2; $b=5; $c=$a*$b; #c=10
/	Divisão	Divide o valor de 2 variáveis	$a=5; $b=2; $c=$a/$b; #$c=2.5
**	Potenciação	Eleva o valor de uma variável por outra	$a=2; $b=3; $c=$a**$b; #$c=8
%	Resto	Retorna o resto da divisão de 2 variáveis	$a=9; $b=7; $c=$a%$b; #$c=2
++	Incremento	Incrementa a variável	$a=1; $a++; #$a=2;
--	Decremento	Decrementa a variável	$a=3; $a--; #$a=2;
+=	Atribuição soma	Soma o valor da segunda variável à primeira	$a=4; $b=3; $a-=$b; #$a=7
-=	Atribuição subrtação	Subtrai o valor da segunda variável da primeira	$a=4; $b=3; $a-=$b; #$a=1

Sintaxe - Operadores

Operador	Nome	Explicação	Exemplo
.	Concatenação	Concatena 2 strings	$nome = “Raissa”; $sobrenome=”Endring” $nomeCompleto=$nome.$sobrenome;
x	Repetição	Concatena uma string Repetidas vezes	$string = “ola”; Print %stringx10;
.=	Atribuição Concatenação	Concatena a segunda String à primeira	$nome=”Janaina”; $sobrenome=”Dantas”; $nome.=$sobrenome;

Sintaxe - Operadores

Operador	Numérica	String
Igual	==	eq
Diferente	!=	ne
Menor	<	lt
Maior	>	gt
Menor ou igual	<=	le
Maior ou igual	>=	ge
Comparação	<=>	cmp

Sintaxe - Controle de Fluxo

- If

```
if ($idade < 18){
        print "Menor de Idade.\n";
}       elsif ($idade<60){
                print "Adulto.\n";
        }
                else{
                        print ("Velho.\n");
        }
```

Sintaxe - Estruturas de Repetição

- While

```
 5 my $i = 0;
 6 while ($i < 10){
 7           print ($i."\n");
 8           $i = $i + 1;
 9
10 }
```

Sintaxe - Estruturas de Repetição

- For

```
6 for ($i=0; $i <= 10; $i++) {
7         print ("Bom dia\n");
8 }
```

Sintaxe - Estruturas de Repetição

- Foreach

```
3 my @nomes = (Janaina, Raissa, Vitor, Leonardo);
4 foreach my $nome_x (@nomes)
5 {
6
7                 print $nome_x . "\n";
8
9 }
10
```

Referência

- Referências são ponteiros para tipos de dados previamente definidos.

```
 3 $num = 7;
 4 @lista = (1,2,3,4,5);
 5
 6 $ref_num = \$num;
 7 $ref_lista = \@lista;
 8
 9 print "$ref_num\n";
10 print "$$ref_num\n";
11
12 print "$ref_lista\n";
13 print "@$ref_lista\n";
```

```
SCALAR(0x17d9ac0)
7
ARRAY(0x17d9b08)
1 2 3 4 5
```

Expressões Regulares

- Expressões Regulares são padrões de procura definidos para caracteres e strings.

- Uma expressão regular (ER) é normalmente designada entre barras (**/**), e seu reconhecimento (*matching*) pode ser feito através do operador lógico "**=~**".

```
 8 if ("Linguagens de Programação" =~ /grama/){
 9     print "Sim\n";
10 }
```

Expressões Regulares

- Pode-se fazer expressões regulares mais genéricas e poderosas, utilizando metacaracteres, que dentro de uma expressão regular têm uma função.

	Função
.	caracter qualquer
[ ]	lista de caracteres permitidos
[^]	lista de caracteres proibidos
?	zero ou um
*	zero ou mais
+	um ou mais
|	um ou outro
\b	borda de palavra
\d	dígito
\n	quebra de linha
\t	tab
{ , }	quantidade de caracteres
\s	espaço, tab ou quebra de linha

Expressões Regulares

Exemplos:

```
12 "OlaAmigos" =~ /Ola\b/; # False
13
14 "Ola!Amigos" =~ /Ola\b/i; # True

17 "Linguagens de Programação" =~ /[axy]/; # True
18
19 "Linguagens de Programação" =~ /a|x|y/; # True

23 "Expressões regulares" =~ /Ep*/; # True
24
25 "Expressões regulares" =~ /Ep+/; # False
```

Expressões Regulares

- Diversas funções de Perl aceitam expressões regulares como argumentos. O exemplo a seguir mostra como dividir as colunas de uma string separada por tabs:

```
@columns = split(/\t/, $string);
```

- Uma expressão é *case-sensitive*, mas, colocando-se *i* no final, faz-se a procura igualando maiúsculas e minúsculas.

```
$string = "DSC00035.JPG";
if ($string =~ /dsc[0-9]+[.]jpg/i)
{
    print "Foi\n";
}
```

Expressões Regulares

- O uso de parênteses permite retornar trechos encontrados na string.

```
 8 if ("22:59" =~ /(\d\d):(\d\d)/){
 9     my $horas = $1;
10     my $minutos = $2;
11     print "$horas $minutos\n";
12 }
```

Sub-rotina

- A definição de um sub-rotina é feita com a sintaxe:

```
3 sub funcao{
4     #
5     # faz o necessário
6     #
7 }
```

- Não é necessária a definição da quantidade e tipo de parâmetros ou de retorno.
- Os parâmetros passados na chamada da sub-rotina ficam armazenadas num array **@_**
- Caso não tenha **return**, o resultado da subrotina é a última expressão avaliada

Sub-rotina

Exemplo:

```
 3 sub maior{
 4     if($_[0]>$_[1]){
 5         $_[0];
 6     }else{
 7         $_[1];
 8     }
 9 }
10
11 $num1 = 5;
12 $num2 = 7;
13
14 $maior = maior($num1,$num2);
15
16 print $maior; # Resultado: 7
```

Arquivos

```
 3 open(ARQ,"entrada.txt") or die "Não foi possivel abrir o arquivo: $!";
 4
 5 while(<ARQ>){
 6     print $_;
 7     push(@lista,$_);
 8 }
 9
10 close(ARQ);
```

“<arquivo.txt” ou “arquvio.txt” aberto para leitura.

“>arquivo.txt” escrita com sobrescrita.

“>>arquivo.txt” escrita sem sobrescrita.

“+>arquivo.txt” leitura e escrita.

Coletor de Lixo

- Perl utiliza um mecanismo bem simples como coletor de lixo.
- Contadores de referência para determinar quando um bloco está pronto para ser coletado. Isso garante que enquanto haja referência para uma informação, ela não será deletada.

Módulos

Um módulo é um programa ou conjunto de programas (.pm) que são utilizados pelos scripts para auxilio em diversas ocasiões.

A palavra reservada “use” carrega módulos no programa.

Alguns exemplos de módulos que alteram o comportamento normal do interpretador:

- strict
- warnings
- diagnostics

Módulos

Você também pode criar seus próprios módulos.

```perl
#Math.pm

package Math

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(add mult);

sub add{
    my $x = $_[0];
    my $y = $_[1];
    return $x + $y;
}

sub mult{
    my $x = $_[0];
    my $y = $_[1];
    return $x*$y;
}

":)";
```

```perl
#Teste.pl

use Math;

my $a = 4;
my $b = 7;

my $soma = add($a,$b);

my $multi = mult($a,$b);
```

Módulos

- CPAN: *Comprehensive Perl Archive Network*
- Maior repositório de módulos para Perl

Polimorfismo

- Coerção: Perl suporta coerção.

```
#Exemplo 1:
my $num1 = "2";
my $num2 = 3;
my $num3 = $num1 + $num2;    #Converte a string "2"
                             #implicitamente para inteiro
                             #e faz a soma
print $num3; #Resposta: 5


#Exemplo 2:
my $tam = @lista; #tam recebe o tamanho da lista


#Exemplo 3:
my $num = 2;
my @lista = $num; # Faz a lista com 1 elemento
```

Polimorfismo

- Sobrecarga: Em perl existe sobrecarga de operadores de uma classe, através do módulo **overload**.

```
3 package Classe;
4
5 use overload
6     "-" => "meuMenos",
7     "+" => \meuMais;
```

- Inclusão: Perl tem polimorfismo de inclusão, como herança entre pacotes, quando usados como classes.

Polimorfismo

- Em outras linguagens de programação o tipo do operando define como a operação irá se comportar. Ao adicionar dois números ocorre uma operação de adição numérica, enquanto que adicionar dois textos acaba por concatená-las. Esta característica chama-se sobrecarga de operador.

Polimorfismo

O Perl de forma geral trabalha de forma diferente.

```
3 $num = 2;
4 $str = " oi";
5
6 $conc = $num.$str;
7 print $conc."\n"; # Imprime "2 oi"
8
9 $ad = $num + $str;
10 print $ad."\n"; # Imprime "2" ($str não é avaliado como valor)
```

Programação Orientada a Objetos

- É possível implementar OO em Perl, porém essa implementação é no mínimo “excêntrica”.
- Em Perl, classes podem ser criadas através de packages.
- As classes são referências para os pacotes, e essa “ligação” é feita através da função *bless.*
- O método construtor é chamado new.
- Há um procedimento padrão no construtor, que deve ser executado invariavelmente.

Programação Orientada a Objetos

```perl
#Pessoa.pm
package Pessoa;

sub new{
    my $class = shift;
    my $self = {
        nome => shift;
        idade => shift;
    }
    bless($self,$class);
    return $self;
}
```

- determinar a classe.
- criar a instância e seus respectivos atributos.
- transformar a instância numa instância real da classe (com o operador bless).
- retornar a instância pronta.

Programação Orientada a Objetos

Agora, para a criação dos métodos.

```perl
sub getNome{
    my $self = shift;    #Pega o objeto que estamos lidando
    return $self->{nome};
}

sub getIdade{
    my $self = shift;
    return $self->{idade};
}

sub aniversario{
    my $self = shift;
    $self->{idade}+=1;
    return $self->{idade};
}

1;
```

Programação Orientada a Objetos

Utilizando a classe criada:

```
#!/usr/bin/perl5.14.2

#teste.pl

use Pessoa;

my $joao = Pessoa->new("João de Souza", 23);

$idadeDoJoao = $joao->getIdade;
print $idadeDoJoao,"\n"; # Resposta: 23

$joao->aniversario; #João fez aniversário :D

$idadeDoJoao = $joao->getIdade;
print $idadeDoJoao,"\n"; # Resposta: 24
```

Tratamento de Exceções

- Assim como C, não é comum se usar o termo “tratamento de exceções” em Perl.

- Perl não possui um mecanismo de tratamento de exceções, muito por conta de ter em seu “background” linguagens como awk, C e shell script, que não possuem mecanismos de exceções.

- Perl inclui facilidades que permitem a implementação de funcionalidades similares (tratamento de erros).

Tratamento de Exceções

- A função **die()** é usada para terminar um programa e apresentar uma mensagem para que o usuário possa ler, que será adicionada na saída de erro padrão STDERR.

- Um exemplo de uso do die() em tratamento de erros:

```
1 #!/usr/bin/perl
2
3 chdir('/user/printer') || die();
4
```

Tratamento de Exceções

- Quando algo de errado acontece em seu programa/script, é de costume alertar o usuário sobre o problema.
- A função **warn()** tem a função de imprimir diretamente na saída de erros padrão STDERR, informando a linha onde ocorreu um determinado erro.

```
3 warn "Slight problem here.";
```

Geraria:

```
erus@CORINGA:~/Área de Trabalho$ perl teste.pl
Slight problem here. at teste.pl line 3.
```

Tratamento de Exceções

- O tratamento de exceções, de um modo geral, não é um fator primordial para os própositos da maioria dos programas feitos em Perl.

- Apesar de não possuir nativamente mecanismos de tratamento de exceções, Perl possui módulos que oferecem recursos de tratamento de exceções de forma similar ao que se é encontrado nas linguagens POO.

- Um exemplo é o módulo **Error.pm**, que implementa os comandos try/catch/throw para tratar exceções.

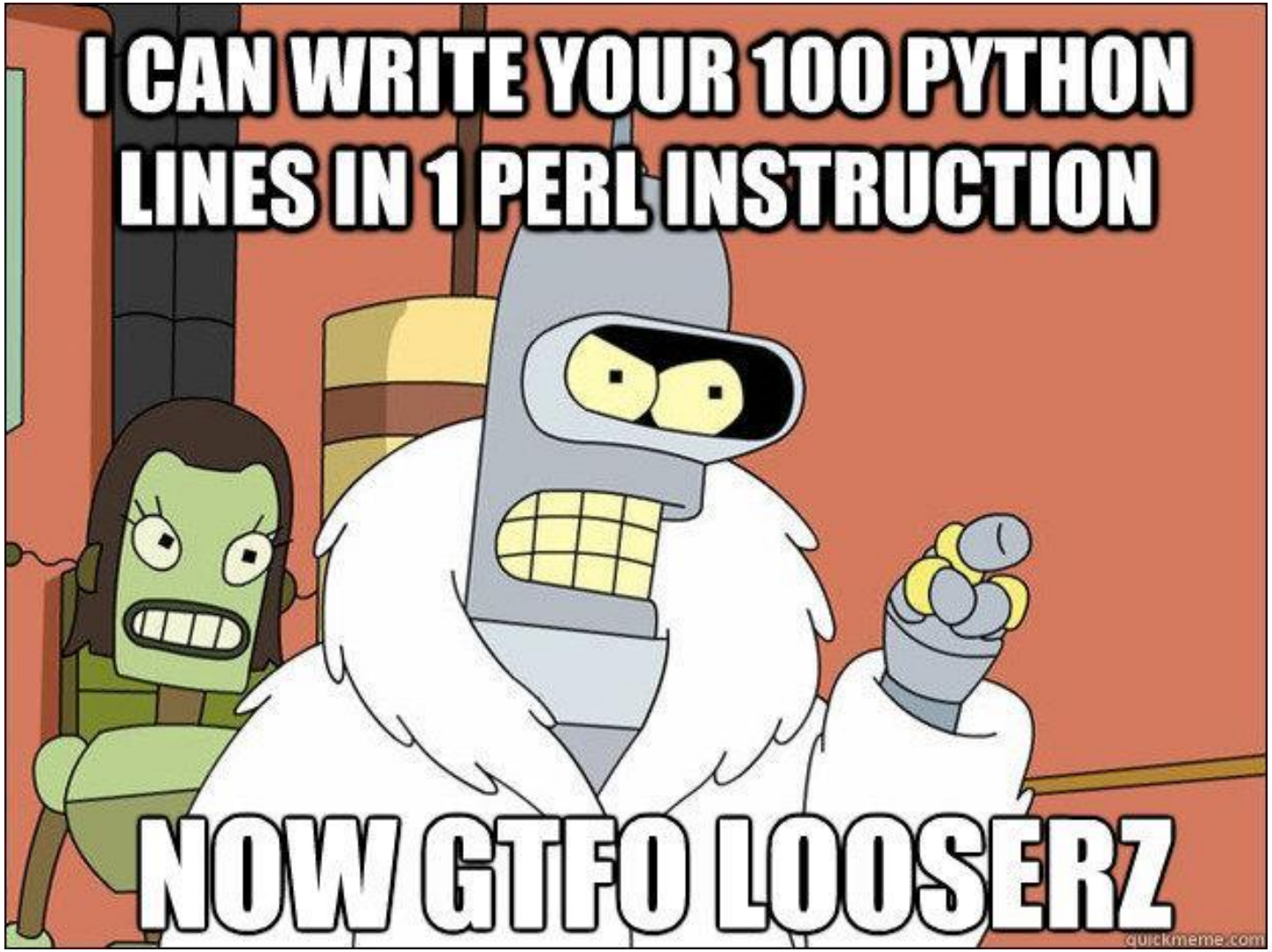
Avaliação da Linguagem

Legibilidade	Não
Redigibilidade	Sim
Aplicabilidade	Sim
Confiabilidade	Não
Facilidade de Aprendizado	Não
Eficiência	Sim
Portabilidade	Sim
Método de Projeto	Estruturado e OO
Evolutibilidade	Sim
Reusabilidade	Sim
Integração	Parcial
Custo	Sim

Avaliação da Linguagem

Verificação de tipos	Dinâmica.
Gerenciamento de Memória	Sistema.
Passagem de Parâmetros	Referência.
Polimorfismo	Coerção, Sobrecarga e Inclusão.
Exceções	Parcial
Concorrência	Parcial

Avaliação da Linguagem



PERL

Referências

- http://perl.org.br/
- http://www.perl.org
- http://perlmaven.org.br
- http://www.perlmonks.org
- http://www.inf.ufes.br/~vitorsouza