++ sobre a linguagem Perl

Assistente de ensino: Marcelo da Silva Reis[1]

Professor: Fabio Kon[1]

[1]Instituto de Matemática e Estatística, Universidade de São Paulo

MAC0211 - Laboratório de Programação I

17 de junho de 2009

IME - Instituto de Matemática e Estatística

Conteúdo (hoje):

"Previously on MAC211..."
- Revisão da aula anterior

Mais sobre expressões regulares
- Casamento (*matching*)
- Substituições
- Exercício

E/S, manipulação de arquivos
- Exercício

Subrotinas
- Sintaxe e exemplos
- Exercício

Depurando códigos em Perl

Conteúdo

Características e Aplicações

- Originalmente utilizada para processamento de textos
  - Várias facilidades para processamento de texto estão "embutidas" na linguagem
- Hoje em dia também utilizada para muitas outras aplicações:
  - administação de sistemas
  - bioinformática
  - aplicações *web*, etc.
- Desenvolvida para ser prática (fácil de usar, eficiente, completa), ao invés de "bela" (elegante, minimal) [1]

[1] fonte: CPAN.org.

Tipos de dados

Os cinco tipos de dados fundamentais em Perl são:

- **escalares:** podem ser números, strings ou referências
- **array:** uma lista ordenada de escalares
- **hash:** um mapeamento de strings para escalares
- **manipulador de arquivo:** um mapeamento para um arquivo ou dispositivo
- **subrotina:** um mapeamento para uma subrotina

Exemplos

Exemplos de declarações, uma variável de cada tipo:

```
my $foo; $bar[$i]; $baz{$k}  # escalar

my @bar;    # um array, default lista vazia

my %baz;    # um hash, default hash vazio

sub marino { ... }    # uma subrotina

# manipuladores de arquivos nao sao declarados:
#
open(ARQ, ... ) # (caixa alta opcional)
```

Operador `split`

O operador `split` devolve uma lista de palavras de um *string*, separadas por um delimitador. Exemplo:

```
@array = split " ", "Foo Bar Baz";

foreach my $valor (@array){
  print $valor . " : ";
}
#
# Imprime "Foo : Bar : Baz : "
#
```

Foreach

O loop `foreach` é amigável para a manipulação de listas armazenadas em *arrays* (e, indiretamente, também de *hashes*):

```
foreach (@meses) {
    print "Mes: $_\n";
}
```

```
# array com 3 elementos
#
print $numeros[$_] foreach 0 .. 2;
```

Exercício 1

Implemente o filtro `wc.c`, tal qual visto em Aula 12 de MAC211. Esse filtro deve ler da entrada padrão um texto e imprimir na saída padrão o número de ocorrências de linhas, de palavras e de caracteres. **Dicas:**

```
$size_of_string = length $string;

while(<STDIN>){ ... }

@array = split /\s+/, $string;
```

Uma solução do Exercício

O filtro "wc.c", da Aula 12, implementado em Perl:

```
my ($nc, $nl, $nw);

while(<STDIN>){
    $nc += length $_;
    $nl++;
    my @palavras = split /\s+/, $_;
    $nw += @palavras;
}

print "$nl, $nw, $nc\n";
```

Conteúdo

Expressões regulares

Expressões regulares têm amplo suporte na linguagem Perl; vamos introduzir algumas operações básicas:

Matching: trata-se da operação mais elementar. Exemplos:

```
if (/foo/){ ... }  # verdadeiro se $_ contem "foo"

if ($tmp =~ /foo/){  # verdadeiro se $tmp contem "foo"
   ...
}
```

Mais *matching*

```
if(/\d+[^\d+]$/){
  #
  # verdadeiro se $_ contem 1 ou mais digitos
  # seguido(s) de um ou mais nao digitos ate'
  # ao final de $_ .
}
```

```
$tmp =~ /^MAC211\s+is\s+.*$/ and print "Ok!";
#
# verdadeiro se $tmp contem "MAC211" no inicio,
# seguido de um ou mais espacos, "is", um ou mais
# espacos e um string de zero ou mais caracteres.
```

Substituições

Substituições são muito úteis na manipulação de strings:

```
s/foo/bar/; # substitui foo por bar em $_


$tmp =~ s/\w+\d+/bar/;
# utiliza metacaracteres para o matching;
# substitui o primeiro matching por bar em $tmp


$tmp =~ s/foo/bar/ig;
# substitui TODOS os foo por bar em $tmp,
# case insensitive

$contador = ($tmp =~ s/for/bar/g);
# conta o total de substituicoes
```

Substituições com captura

Podemos capturar parte do *matching* da substituição e utilizá-lo para definir a expressão a ser trocada:

```
$_ = "Foo Bar Baz";

s/(\w+)/$1tball/; #  $_ eq "Football Bar Baz"

$k = (s/(b)(a)/$2$1/ig);

#  $_ eq "Footabll aBr aBz"
#
#  $k == 3
```

Operador `tr`

O operador de transliteração (`tr`) também serve para modificar *strings*

```
$string =~ tr/[a-z]/[A-Z]/; # mesmo que 'uc $string'
```

Operador `tr`

O operador de transliteração (`tr`) também serve para modificar *strings*

```
$string =~ tr/[a-z]/[A-Z]/; # mesmo que 'uc $string'
```

Exemplo:

```
$DNA1 = "ATGCCTA";   # quero o complemento reverso!

$DNA1 =~ tr/ACGT/TGCA/;

$DNA2 = reverse $DNA1;  # $DNA2 eq "TAGGCAT"
```

Filtragem (captura)

Outra coisa legal é a filtragem de textos utilizando expressões regulares. Exemplo de filtragem dos campos de um email:

```
if ($email =~ /([^\@]+)\@(.+)/) {
        print "Nome: $1\n";
        print "Dominio: $2\n";
}
```

As expressões capturadas são definidas pelos parênteses, sendo armazenadas nas variáveis $1, $2, etc.

Mais exemplos de filtragem

```
$frase = "Use the Fork!";

$frase =~ /^Use\s+the\s+(.+)/
     and print "Dont use the " . $1;
#
# Verdadeiro, filtra "Fork!"
```

Sempre é bom testar se houve *matching* ao proceder a filtragem, pois em caso negativo todas as variáveis `$1`, `$2`, etc. ficam setadas como `undef`.

Função `join`

A função `join` recebe um string e um "agrupador" e devolve um *string*. Exemplo:

```
@array = ("Foo", "Bar", "Baz");

@string = join " : ", @array;

print $string;
#
# Imprime "Foo : Bar : Baz"
#
```

Variáveis especiais de *matching*

Além de `$1`, `$2`, etc., quando há um casamento de uma expressão três variáveis especiais são setadas:

- `$‘` : recebe o string à esquerda do *matching*
- `$&` : recebe o *matching*
- `$’` : recebe o string à direita do *matching*

Teste de expressões regulares (*Learning Perl, 8*)

O seguinte programa é útil ao elaborarmos expressões regulares em Perl:

```
#!/usr/bin/perl
while (<STDIN>) {     # take one input line at a time
  chomp;
  if (/YOUR_PATTERN_GOES_HERE/) {
    print "Matched: |$`<$&>$'|\n";
  } else {
    print "No match: |$_|\n";
  }
}
```

Exercício 2 (*Learning Perl* 8.6)

Escreva um programa que lê da entrada padrão e verifica, para cada linha, se a mesma tem espaços em branco antes do *newline*. Se existir, acrescente uma cerquilha (#) ao final da linha. As linhas, modificadas ou não, devem ser impressas na saída padrão. **Dicas:**

```
while(<STDIN>){ ... }

chomp $_;  # remove o '\n'

$string =~ /YOUR_PATTERN_GOES_HERE/;
```

Conteúdo

Manipulando arquivos

A manipulação de arquivos em Perl é simples:

```
open(my ARQ,  "<" ,  "arquivo.txt" );

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

close(ARQ);
```

Tratando exceções

O trecho de código:

```
open(IN, "<", "input.txt") or die "Erro!";
```

Equivale a:

```
if(!open(IN, "<" "input.txt")){
    print "Erro!";
    exit 1;
}
```

print e prinft

Uma observação sobre o print:

```
print "'Hallo!'\n";
# imprime "'Hallo!'", com quebra de linha

print 'Hallo, $string!\n';
# imprime "Hallo, $string!\n", sem quebra de linha
```

print e prinft

Uma observação sobre o print:

```
print "'Hallo!'\n";
# imprime "'Hallo!'", com quebra de linha

print 'Hallo, $string!\n';
# imprime "Hallo, $string!\n", sem quebra de linha
```

O printf funciona de forma muito similar ao de C:

```
my $pi = 3.14;
my $pi_name = "Pi";

printf "%s value is %1.2f", $pi_name, $pi;
#
# imprime "Pi value is 3.14"
```

prinft e arquivos

Podemos utilizar o prinft para escrever em arquivos:

```
my $pi = 3.14;
my $pi_name = "Pi";

printf ARQ "%s value is %1.2f", $pi_name, $pi;
#
# imprime em ARQ "Pi value is 3.14"
```

Input, output, append

```
open(my IN,  "<" ,  "input.txt" ) or die "Erro!\n";
open(my OUT, ">" ,  "output.txt") or die "Erro!\n";
open(my APP, ">>",  "append.txt") or die "Erro!\n";

printf OUT "Escrevendo no arquivo output.txt\n";

while(<IN>){
    print $_;
    printf OUT $_;
    printf APP $_;
}

close(IN);
close(OUT);
close(APP);
```

Tratando argumentos

Os argumentos são armazenados na variável de ambiente `@ARGV`

Tratando argumentos

Os argumentos são armazenados na variável de ambiente `@ARGV`

`scalar(@ARGV)` é o número de argumentos

Tratando argumentos

Os argumentos são armazenados na variável de ambiente `@ARGV`

`scalar(@ARGV)` é o número de argumentos

O nome do programa é armazenado em `$0`

Tratando argumentos

Os argumentos são armazenados na variável de ambiente `@ARGV`

`scalar(@ARGV)` é o número de argumentos

O nome do programa é armazenado em `$0`

Exemplo:

```
print "Recebi de $0 " . @ARGV . "argumentos:\n";
foreach(@ARGV){
    print $ARGV[$_] . " ";
}
```

```
bash$>perl -w teste.pl um dois
Recebi de teste.pl 2 argumentos:
um dois
```

O operador *diamond*

O operador *diamond* (<>) é muito útil para o processamento de vários arquivos ao mesmo tempo:

```
while(<>){
    print $_;
#
# imprime todas as linhas de $ARGV[0], depois
# todas as linhas de $ARGV[1], etc.
#
}
```

Processando múltiplos arquivos

Se um ou mais arquivos passados como argumento não existir(em), é exibida um aviso e a execução do programa segue adiante.

Processando múltiplos arquivos

Se um ou mais arquivos passados como argumento não existir(em), é exibida um aviso e a execução do programa segue adiante.

Se nenhum argumento é passado (ou seja, se `@ARGV == 0`), então

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

e

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

São equivalentes.

Exercício 3

Escreva um filtro que lê uma sequência de arquivos-texto cujos nomes são passados como argumentos e joga na saída padrão a frequência relativa de cada uma das 26 letras do alfabeto no conjunto formado pela concatenação desses arquivos. Em seguida, imprima, na saída padrão, uma mensagem indicando qual a vogal e qual a consoante mais frequente nos textos fornecidos.

Caso nenhum argumento seja passado, ele deve ler o texto da entrada padrão.

Dicas do exercício 3

```
while(<>){ ... }

# O 'sort' devolve uma lista de chaves do hash ordenadas
# numericamente de acordo com os valores.
#
@array = sort {$hash{$a} <=> $hash{$b}} keys %hash;

$maiusculas = uc $string;

$minusculas = lc $string;

foreach ('A'..'Z'){
    ...
}
```

Conteúdo

Um exemplo simples de subrotina

```
sub logger {
  my $mensagem = $_[0];
  open my LOG, ">>", "meu.log" or die "Erro!";
  print $logfile $logmessage;
}

&logger "Teste de subrotina";
```

Como toda variável em Perl, subrotinas podem ser declaradas em qualquer parte do código!

Um exemplo simples de subrotina

```
sub logger {
  my $mensagem = $_[0];
  open my LOG, ">>", "meu.log" or die "Erro!";
  print $logfile $logmessage;
}

&logger "Teste de subrotina";
```

Como toda variável em Perl, subrotinas podem ser declaradas em qualquer parte do código!

Porém, é uma boa prática declarar todas no início ou no final

Um exemplo simples de subrotina

```
sub logger {
  my $mensagem = $_[0];
  open my LOG, ">>", "meu.log" or die "Erro!";
  print $logfile $logmessage;
}

&logger "Teste de subrotina";
```

Como toda variável em Perl, subrotinas podem ser declaradas em qualquer parte do código!

Porém, é uma boa prática declarar todas no início ou no final

O & pode ser omitido caso não tenha ambiguidades

Passando parâmetros

Um exemplo simples de chamada de uma função:

```
imprime_alunos (\%idade, @nome);
```

No exemplo acima, o primeiro parâmetro é passado por referência, enquanto o último é passado por valor.

```
sub imprime_alunos {
  my ($idade, @nome) = @_;
  print "$nome[$_] $idade->{$nome->[$_]}"
                    foreach 0..$#nome;
}
```

Os argumentos da subrotina são armazenados no array `@_`

Devolvendo valores

Podemos devolver escalares ou listas:

```
sub teste {
  return (1, 2, 3);
}
```

O `return` pode ser omitido:

```
sub marino {
  my $teste = shift;  # shift @_;
  if ($teste){
    return "foo";
  }
  undef;   # mesma coisa que "return undef"
}

my $tem_foo = marino($variavel);
```

Exercício 4

Elabore uma subrotina em Perl que receba uma lista de nomes, dois *hashes* de notas de provas (cujas chaves são os nomes da lista) e devolva uma lista com a média aritmética dos alunos.

Ferramentas:

```
$c = $a / $b;

$ref_hash = \%hash;

sub minha_subrotina{
  my ($x, $y, $z) = @_;
  ...
  return $resultado;   # ou @resultado
}

foreach my $chave (keys %hash){ ... $hash{$chave} ... }
```

IME - Instituto de Matemática e Estatística

Conteúdo

Corrigindo programas em Perl

- Antes de tudo, muitos problemas são evitados utilizando a flag `-w` quando se executa um código Perl
  - Utilizar o `use warnings;` tem o mesmo efeito
  - O `use strict;` também é útil para prevenir erros
  - Executar `perl -c meu_codigo.pl` verifica sintaxe e typos

Corrigindo programas em Perl

- Antes de tudo, muitos problemas são evitados utilizando a flag `-w` quando se executa um código Perl
  - Utilizar o `use warnings;` tem o mesmo efeito
  - O `use strict;` também é útil para prevenir erros
  - Executar `perl -c meu_codigo.pl` verifica sintaxe e typos
- Feito isso, é comum corrigir código Perl utilizando o "comentar e imprimir"

Corrigindo programas em Perl

- Antes de tudo, muitos problemas são evitados utilizando a flag `-w` quando se executa um código Perl
  - Utilizar o `use warnings;` tem o mesmo efeito
  - O `use strict;` também é útil para prevenir erros
  - Executar `perl -c meu_codigo.pl` verifica sintaxe e typos
- Feito isso, é comum corrigir código Perl utilizando o "comentar e imprimir"
- Se os passos acima não resolverem, podemos utilizar o depurador Perl

Iniciando o depurador Perl

- Para iniciar o depurador Perl, basta digitarmos na linha de comando:

```
bash$ perl -d programa_bixado.pl
```

- Alternativamente, podemos modificar a primeira linha do código:

```
#!/usr/bin/perl -d
```

Alguns comandos básicos do depurador

- `h` : exibe uma tela de ajuda (`h h` para ajuda detalhada)
- `b n` : define um breakpoint na linha n (ex: `b 42`)
- `p var` : imprime o estado de uma variável (ex: `p $pe`)
- `r` : inicia a execução (`R` para reiniciar uma)
- `q` : sai do depurador

Referências

1. Perl.org. `http://www.perl.org/`.
   Acesso em 17 de junho de 2010.

2. Comprehensive Perl Archive Network.
   `http://www.cpan.org/`.
   Acesso em 17 de junho de 2010.

3. Livros da O'Reilly:
   - *Learning Perl*.
   - *Programming Perl*.

4. Verbete "Foo bar" na Wikipédia.
   `http://en.wikipedia.org/wiki/Foo_bar`.
   Acesso em 17 de junho de 2010.