CGI/Perl: O Início Perl é uma linguagem de programação, e CGI é um protocolo de comunicação (um código) através do qual o servidor HTTP (ou servidor web) intermedia a transferência de informações entre um programa (no mesmo computador que o servidor) e um cliente HTTP (o seu browser).
Não é correto falar "eu quero um CGI que faça tal coisa", mas sim "eu quero um programa que use CGI e faça tal coisa", ou de maneira mais curta "eu quero um programa CGI que faça tal coisa" ou "eu quero um script que faça tal coisa".
Um script não implica em CGI, mas scripts Perl (programas Perl) tornaram-se tão comuns em se tratando de CGI, que falar script já quer dizer "programa CGI escrito em Perl".
É sempre bom lembrar que um programa CGI pode ser feito em qualquer linguagem usada no mesmo computador que o servidor, não só em Perl.
Mas o status alcançado por Perl é devido a suas qualidades para tarefas associados a CGI e por sua portabilidade.

1. Um programa simples
    - A primeira linha
    - Comentários e declarações
    - Imprimindo na tela

2. Executando o programa

3. Variáveis escalares
    - Operações e Atribuições
    - Interpletação
    - Exercício

4. Variáveis de matrizes
    - Atribuições de matrizes
    - Mostrando matrizes
    - Exercício

5. Manipulando arquivos
    - Exercício

6. Estruturas de controle
    - foreach
    - Testando
    - for
    - while e until
    - Exercício

7. Condicionais
    - Exercício

8. Comparando strings
    - Expressões regulares
    - A variável especial $_
    - Mais sobre ERs
    - Alguns exemplos de ERs
    - Exercício

9. Substituição e tradução
    - Opções
    - Recordando modelos
    - Tradução
    - Exercícios

10. Split
- Exercício

11. Matrizes associativas
- Operadores
- Variavéis de Ambiente

12. Subrotinas
- Parâmetros
- Retornado Valores
- Variáveis Locais

Um programa simples Aqui está um programa simples que usaremos para começarmos:

```
#!/usr/local/bin/perl
print 'Olá mundo.'; # mostra uma mensagem
```

Cada parte será discutida a seguir.

A primeira linha Todo programa escrito em Perl deve sempre começar com esta linha:

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

Contudo, ela pode variar de sistema para sistema. Esta linha indica para o servidor o que fazer com o arquivo quando ele é executado (i.é, rodar o programa através do Perl) e informa a localização exata do compilador.

Comentários e declarações Comentários podem ser inseridos em um programa através do símbolo `#`, e qualquer coisa após ele, até o final da linha, é ignorado (com exceção da primeira linha). A única maneira de usar comentários longos é colocando `#` no início de cada linha.
Toda declaração em Perl deve terminar com um ponto-e-vírgula, como na última linha do programa acima.

Imprimindo na tela A primeira função exibe informações. No caso acima, ele mostra a cadeia de caracteres Olá mundo.
Este resultado será mostrado somente se for executado na linha de comandos. Para que a mesma informação seja mostrada pelo seu browser, você deve modificar o programa, de forma que a mensagem seja reconhecida pelo formato HTML ou como texto simples.

```
print "Content-type: text/html\n\n"; # informa ao browser

print 'Olá mundo.';   # mostra a mensagem simples, sem formatação
```

O próximo passo é rodá-lo.

Executando o programa Digite o programa acima em um editor de texto, depois salve ele. Existem duas formas de instalar seu programa no servidor:
Pelo Unix (requer conhecimento básico), Emacs é um bom editor porque possui o modo Perl que formata linhas facilmente quando você pressiona tab (use M-x perl-mode).
Pelo FTP, use qualquer editor de texto puro, e envie o arquivo através de um programa de FTP (CuteFTP ou WS_FTP), é importante que seja usado o modo de transferência ASCII, e nunca Binário.
O programa deve ser salvo com a extensão .cgi (que indica ser um script) ou com .pl (de Perl), e depois torná-lo executável.
Pelo Unix, digite:
chmod u+x programa

na linha de comandos, onde programa é o nome do seu arquivo.
Pelo FTP, clique em alterar atributos do arquivo, escolhendo o valor 755 (-rwxr-xr-x).
Agora rode o programa apenas digitando um destes comandos:
perl programa (pelo DOS ou UNIX)
./programa (pelo UNIX)
programa (pelo DOS)
Caso você esteja usando a linha de comandos do DOS, é necessário ter o programa Perl for Win32 instalado, ele pode ser retirado no endereço:
http://www.activestate.com/
Se algo ocorrer de errado, podem aparecer mensagens de erro, ou nada. Você pode sempre rodar o programa com mensagens através do comando:
perl -w programa
Ele mostrará avisos e outras mensagens de ajuda antes de tentar executar o programa. Para rodá-lo com a depuração, use o comando:
perl -d programa
O Perl compila primeiro o programa e então executa sua versão compilada. Assim, após uma pequena pausa para compilação, o programa deve rodar rapidamente.
Certifique-se de que o programa está funcionando antes de continuar.

Variáveis escalares O tipo mais usado de variável no Perl é o escalar. Variáveis escalares podem conter caracteres e/ou números, e note que eles são completamente intercambiáveis. Por exemplo, a declaração:
```
$prioridade = 9;
```
atribui a variável escalar $prioridade o valor 9, mas você pode também designar uma string (cadeia de caracteres) para a mesma variável:
$prioridade = 'alta';
Perl também aceita números como strings:
```
$prioridade = '9';
$default = '0009';
```
e ainda pode realizar operações aritméticas ou outras.
Em geral, nomes de variáveis consistem de números, letras e símbolos _ mas não devem começar com números depois do `$`. A variável `$_` é especial, como veremos mais tarde. Além disso, o Perl é case sensitive, i.é, letras maiúsculas são diferentes de minúsculas, por exemplo, `$a` e `$A` são variáveis distintas.
Operações e Atribuições Interpletação Exercício

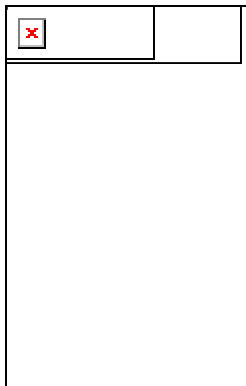
Operações e Atribuições O Perl utiliza todas as operações usuais na linguagem C:
```
$a = 1 + 2; # soma 1 e 2 e armazena em $a
$a = 3 - 4; # subtrai 4 de 3 e armazena em $a
$a = 5 * 6; # multiplica 5 por 6
$a = 7 / 8; # divide 7 por 8 e retorna 0.875
$a = 9 ** 10; # 9 elevado por 10
$a = 5 % 2; # resto da divisão de 5 por 2
++$a; # incrementa $a e retorna seu valor
$a++; # retorna $a e depois incrementa em 1
--$a; # decrementa $a e retorna seu valor
$a--; # retorna $a e depois decrementa em 1
```
e para caracteres, existem os seguintes operadores:
```
$a = $b . $c; # concatena $b e $c
$a = $b x $c; # $b repetido $c vezes
```
Para atribuir valores, Perl utiliza:
```
$a = $b; # atribui $b para $a
$a += $b; # soma $b para $a
$a -= $b; # subtrai $b de $a
$a .= $b; # acrescenta $b em $a
```

Note que quando Perl atribui um valor como $a = $b, ele faz uma cópia de $b e então o atribui para $a. Portanto, na próxima vez que você alterar $b, ele não irá alterar $a.
Outras operações podem ser encontradas através do comando man perlop na linha de comandos do Unix.

Interpletação O seguinte código mostra maçãs e pêras usando concatenação:
```
$a = 'maçãs';
$b = 'pêras';
print $a.' e '.$b;
```
Seria mais fácil incluir somente uma string no final da declaração print, mas a linha:
```
print '$a e $b';
```
mostra somente $a e $b, e não maçãs e pêras, o que não é o nosso caso. Ao invés disso, podemos usar aspas duplas no lugar de aspas simples:
```
print "$a e $b";
```
Estas aspas forçam a interpolação de qualquer código, incluindo interpretação de variáveis. Isso é mais apropriado que nossa declaração original. Outros códigos que são interpolados incluem caracteres como newline e tab. O código **\n** é o newline (enter/return) e **\t** é o tab.
Outros exemplos que produzem o mesmo resultado:
```
print ' Meu e-mail é nome@dominio.com '."\n";
print " Meu e-mail é nome\@dominio.com \n";
```
Note que, com aspas duplas, se faz necessário o uso da barra invertida antes do @, porque senão ele será interpretado como sendo uma variável, que veremos adiante.

Exercícios Neste exercício você deve reescrever seu primeiro programa de modo que **(1)** a string seja atribuída para uma variável e **(2)** esta variável seja impressa com um caractere newline. Use as aspas duplas e não utilize o operador de concatenação.
Certifique-se de que o programa funciona, antes de prosseguir.

Variáveis de matrizes Um tipo mais interessante de variável é o de matrizes (vetores) que é uma lista de variáveis escalares. Variáveis de matrizes tem o mesmo formato das escalares exceto que eles são prefixados pelo símbolo @. As declarações:
```
@comida = ("maçãs", "pêras", "uvas");
@musica = ("flauta", "gaita");
```
atribuem três elementos para a matriz @comida e dois para a matriz @musica.
A matriz é acessada pelo uso de índices começando do zero, e colchetes são usados para especificar cada índice. A expressão:
```
$comida[2]
```
retorna uvas. Note que o símbolo @ foi mudado para $ porque uvas é o valor de uma variável escalar.
Atribuições de matrizes Mostrando matrizes Exercício



Atribuições de matrizes

No Perl, a mesma expressão em um contexto diferente pode produzir um resultado diferente. A primeira atribuição abaixo expande a variável @musica de modo que fique equivalente à segunda atribuição:

```
@mais_musica = ("órgão", @musica, "harpa");
@mais_musica = ("órgão", "flauta", "gaita", "harpa");
```

Isto deve sugerir um modo de acrescentar elementos em uma matriz. Um modo simples de acrescentar estes elementos é usar a declaração:

```
push(@comida, "ovos");
```

que acrescenta ovos ao fim da matriz @comida. Para acrescentar dois ou mais itens, use um dos seguintes formatos:

```
push(@comida, "ovos", "carne");
push(@comida, ("ovos", "carne"));
push(@comida, @mais_comida);
```

Além disso, a função push retorna o comprimento da nova lista.
Para remover o último item de uma lista e retorná-lo, use a função pop. De nossa primeira lista, a função pop retorna uvas e @comida passa a ter dois elementos:

```
$outra_comida = pop(@comida);# agora $outra_comida =
"uvas"
```

Também é possível atribuir uma matriz para uma variável escalar. A linha:

```
$c = @comida;
```

atribui apenas o comprimento de @comida, mas

```
$c = "@comida";
```

transforma a lista em uma string com um espaço entre cada elemento. Este espaço pode ser substituído por qualquer outro caractere apenas mudando o valor da variável especial $" (por exemplo, $" = " mais ";). Esta variável é apenas uma de muitas outras usadas na linguagem Perl.
Matrizes também podem ser usadas para fazer múltiplas atribuições para variáveis escalares:

```
($a, $b) = ($c, $d); # é o mesmo que $a=$c; $b=$d;
($a, $b) = @comida; # $a e $b são os primeiros itens de
@comida.
($a, @alguma_comida) = @comida; # $a é o primeiro item
de @comida e @alguma_comida é uma lista dos
# demais elementos.
(@alguma_comida, $a) = @comida; # @alguma_comida =
@comida e $a é indefinida.
```

A última atribuição ocorre desta maneira porque matrizes consomem tudo, e @alguma_comida retira todos os elementos de @comida. Portanto, este formato deve ser evitado.
Finalmente, você pode querer encontrar o índice do último elemento de uma lista.
Para fazer isso, use a expressão:

```
$#comida
```

Copyright © 1999 STI Internet. Todos os direitos reservados.

Mostrando matrizes Desde que contexto é importante, não é surpresa que os seguintes comandos produzem resultados diferentes:

```
print @comida; # resultado óbvio
print "@comida"; # tratado como string
print @comida.""; # em um contexto escalar
```

Exercícios Experimente cada uma das três declarações citadas em "Mostrando matrizes" para ver o que elas fazem.

Manipulando arquivos

Aqui está um programa básico em Perl que faz o mesmo que o comando cat do Unix em um certo arquivo:

```
#!/usr/local/bin/perl
#
# Programa para abrir o arquivo de senhas, ler,
# escrever, e fechá-lo.
$arquivo = '/etc/passwd'; # nome do arquivo
open(INFO, $arquivo); # abre o arquivo
@linhas = ; # coloca ele em uma matriz
close(INFO); # fecha o arquivo
print @linhas; # exibe a matriz
```

A função open abre um arquivo para entrada (i.é, para leitura). O primeiro parâmetro é o nome que permite ao Perl referir o arquivo futuramente. O segundo parâmetro é a expressão denotando o nome do arquivo com sua localização.
Se o nome do arquivo for escrito entre aspas, então ele é tomado literalmente sem a expansão shell do Unix. Assim, a expressão '~/etc/passwd' não será interpretada com sucesso. Se você deseja forçar a expansão shell, então use os símbolos < e >, neste caso, <~/etc/passwd>.
A função close diz ao Perl para fechar o arquivo.
Existem poucos pontos para acrescentar sobre a manipulação de arquivos.
Primeiro, a declaração open pode também especificar um arquivo para saída ou para acréscimo, assim como para entrada. Para fazer isso, use o símbolo > para saída e >> para acréscimos:

```
open(INFO, $arquivo); # abre para leitura
open(INFO, ">$arquivo"); # abre para escrita
open(INFO, ">>$arquivo"); # permite acrescentar
open(INFO, "<$arquivo"); # também abre para leitura
```

Segundo, se você quiser gravar alguma informação em um arquivo, você deve abri-lo para escrita e então pode usar a declaração print com um parâmetro extra. Para gravar uma string no arquivo INFO use:

```
print INFO "Esta linha vai para o arquivo.\n";
```

Terceiro, você pode usar o seguinte comando para abrir a entrada padrão (usualmente o teclado) e a saída padrão (usualmente a tela) respectivamente:

```
open(INFO, '-'); # abre entrada padrão
open(INFO, '>-'); # abre saída padrão
```

No programa acima, a informação é lida do arquivo. O arquivo chama-se INFO e para ler ele através do Perl, coloque entre os símbolos < e >.
Assim, a declaração:

```
@linhas = <INFO>;
```

lê o arquivo denotado por INFO e coloca dentro da matriz @linhas.
Note que a expressão <INFO> lê o arquivo inteiro em apenas um

passo. Isto ocorre porque a leitura é tratada como uma variável de matriz.
Se @linhas fosse substituída por uma variável escalar $linhas, então somente a primeira linha seria lida (e depois as próximas). Em ambos casos, cada linha é armazenada completamente com seu caractere newline no final.
Exercício

Exercício

```
#!/usr/local/bin/perl
#
# Programa para abrir o arquivo de senhas, ler,
# escrever, e fechá-lo.
$arquivo = '/etc/passwd'; # nome do arquivo
open(INFO, $arquivo); # abre o arquivo
@linhas = ; # coloca ele em uma matriz
close(INFO); # fecha o arquivo
print @linhas; # exibe a matriz
```

Modifique o programa acima de modo que todo o arquivo contenha o símbolo `#` no início de cada linha.
Você deverá somente acrescentar uma linha e modificar uma delas.
Use a variável `$"`.
Coisas inesperadas podem ocorrer com arquivos, então você pode encontrar alguma ajuda com o uso da opção –w, como mencionado no tópico Executando o programa.

Estruturas de controle Possibilidades mais interessantes aparecem quando nós introduzimos as estruturas de controle e o looping. Perl suporta muitos tipos diferentes de estruturas de controle que tendem a ser como na linguagem C, mas muitos são similares aos do Pascal, também. Aqui discutiremos um pouco delas.
foreach Testando for while e until Exercício

foreach Para executar cada linha de uma matriz ou de outra estrutura com formato de lista (como linhas em um arquivo), Perl usa a estrutura foreach. Ela tem o seguinte formato:

```
foreach $petisco (@comida) # visitar cada item e colocá-lo em $petisco
{
print "$petisco\n"; # exibe cada item
print "Yum yum\n"; # de @comida...
}
```

As ações a serem realizadas cada vez, estão no bloco entre chaves. Na primeira passagem pelo bloco, $petisco é atribuído ao valor do primeiro item da matriz @comida. Na próxima vez, ele é atribuído ao segundo valor, e assim por diante. Se @comida estiver vazio logo no início, então o bloco de declarações não será executado.

Testando As próximas e poucas estruturas contam com os testes booleanos verdadeiro ou falso. O número zero, sendo zero um caractere, e o caractere vazio ("") são contados como falsos. Aqui estão alguns testes em números e strings:

```
$a == $b # $a é numericamente igual a $b? cuidado: não use só o operador = $a
!= $b $a é numericamente diferente de $b?
```

```
$a eq $b # $a é uma string igual a $b?
$a ne $b # $a é uma string diferente de $b?
```

Você pode também tentar usar operadores lógicos e, ou, não:

```
($a && $b) # $a e $b são verdadeiros?
($a || $b) # $a ou $b é verdadeiro?
!($a) # $a é falso?
```

for Perl possui a estrutura for que funciona da mesma forma que no C. Ele tem o formato:

```
for (inicializar; testar; incrementar)
{
primeira_ação;
segunda_ação;
etc;
}
```

Primeiramente, a declaração inicializar é executada. Então, enquanto testar for verdadeiro, o bloco de ações será executado. Após cada passagem, é a vez de incrementar. Aqui está um exemplo de loop para mostrar os números de 0 a 9:

```
for ($i = 0; $i < 10; ++$i)
# começa com $i = 0
# executa enquanto $i < 10
# incrementa $i antes de repetir
# neste caso $i++ tem o mesmo efeito
{
print "$i\n";
}
```

while e until Aqui está um programa que lê do teclado e não continua até que a senha correta seja digitada:

```
#!/usr/local/bin/perl
print "Senha? "; # pede pela senha
$a = <STDIN>; # lê do teclado
chop $a; # remove o newline no final da linha
while ($a ne "teste") # enquanto a senha estiver errada...
{
print " errada.\n Senha? "; # solicita novamente
$a = <STDIN>; # lê do teclado
chop $a; # remove o caractere newline no final
}
```

O bloco de código entre chaves é executado enquanto a entrada não for igual a senha. A estrutura while deve ser de fácil compreensão, mas é oportuno notar diversos detalhes.
Primeiro, nós podemos ler de uma entrada padrão (teclado) sem precisar abri-la.
Segundo, quando a senha é digitada, $a inclui o caractere newline no final (ao pressionar enter/return). A função chop remove o último caractere de uma string que neste caso é o newline.
Para testar o oposto, nós podemos usar a declaração until do mesmo modo. Este executa o bloco repetidamente até que a expressão seja verdadeira, e não enquanto ela é verdadeira.
Outra técnica útil, é colocar o while ou until no final do bloco de declarações, em vez de colocar no início. Isto requer a presença do operador do para marcar o início do bloco e o teste no final.
Novamente, o programa acima pode então ser escrito desta forma (note que o bloco será executado pelo menos uma vez):

```
#!/usr/local/bin/perl
do
{
print "Senha? "; # pede pela senha
$a = ; # lê do teclado
chop $a; # remove o último caractere, newline
}
```

```
while ($a ne "teste");
# repete enquanto a senha for errada
# usando until no lugar de while,
# until ($a eq "teste"); repete até que a senha esteja correta
```

Exercício Modifique o programa do exercício anterior de modo que cada linha do arquivo seja lida uma a uma, e retorne com um número de linha no início. Você deve ter algo como:

```
1 root:oYpYXm/qRO6N2:0:0:Super-User:/:/bin/csh
2 sysadm:*:0:0:System V
Administration:/usr/admin:/bin/shdiag:*:0:996:Hardware
Diagnostics:/usr/diags:/bin/csh
3 etc
```

Talvez seja interessante usar esta estrutura:

```
while ($linha = ) # lê cada linha até o final de INFO
{
...
}
```

Quando você tiver feito isso, veja se pode alterá-lo de modo que cada número de linha seja mostrada como 001, 002, ..., 009, 010, 011, 012, etc.
Para fazer isso, você deverá somente mudar uma linha inserindo quatro caracteres extras. Lembre-se que o Perl permite isso.

Condicionais

É claro que o Perl também permite testes condicionais if/then/else. Eles possuem o seguinte formato:

```
if ($a)
{
print "A variável não está vazia\n";
}
else
{
print "A variável está vazia\n";
}
```

Para entender, recorde que o caractere vazio é considerado como falso. Ele também retorna falso se o caractere for zero.
Também é possível incluir mais alternativas em uma declaração condicional:

```
if (!$a) # ! é o operador não
{
print "A variável está vazia\n";
}
elsif (length($a) == 1) # se acima falir, tente isso
{
print "A variável tem um caractere\n";
}
else # se também falir...
{
print "A variável tem vários caracteres\n";
}
```

Neste caso, é importante notar que a declaração elsif está correta, sem o e de elseif.
Exercício

Exercício Encontre um arquivo grande com algum texto e algumas linhas em branco.
Do exercício anterior, você deve ter um programa que mostra o arquivo de senhas com números de linhas. Mude ele de modo que funcione com o outro arquivo de texto.
Agora altere o programa de modo que aqueles números não sejam mostrados ou contados com linhas em branco, mas toda linha permanece sendo mostrada, incluindo as em branco. Recorde que, quando uma linha do arquivo é lida, ela continua incluindo seu caractere newline no final.

Comparando strings Um dos mais úteis recursos do Perl (senão o mais útil) é a poderosa manipulação de strings. No coração desta, está a expressão regular (ER) que é compartilhada por muitos outros utilitários do Unix.
Expressões regulares A variável especial $_ Mais sobre ERs Alguns exemplos de ERs Exercício

Expressões regulares Uma expressão regular sempre está contida entre barras, e a comparação ocorre com o operador =~. A seguinte expressão é verdadeira se a string aparecer na variável $sentença:

```
$sentenca =~ /para/
```

A ER é case sensitive, de modo que se:

```
$sentenca = "Para as raposas";
```

então a comparação resultará em falsa. O operador **!~** é usado para o oposto, sendo assim o exemplo:

```
$sentenca !~ /para/
```

retorna verdadeiro porque a string não aparece em $sentença

A variável especial $_ Nós podemos usar uma condicional como:

```
if ($sentenca =~ /sob/)
{
print "Nós estamos conversando\n";
}
```

que deverá mostrar a mensagem se tivermos um dos seguintes valores:

```
$sentenca = "Embaixo e sob";
$sentenca = "É sobre o assunto...";
```

Porém é mais frequentemente e muito mais fácil se atribuirmos a sentença à uma variável especial **$_** (certamente escalar).
Se nós fizermos isso, então podemos evitar o uso dos operadores de comparação e o exemplo acima pode ser escrito como:

```
if (/sob/)
{
print "Nós estamos conversando\n";
}
```

A variável **$_** é padrão para muitas operações do Perl e tende a ser muito usado.

Mais sobre ERs Em uma ER existe uma grande quantidade de caracteres especiais, o que faz com que pareça mais complicado. O melhor modo de construir seus programas é usando ERs aos poucos.
Aqui estão alguns caracteres especiais ER e seus significados:

```
. # qualquer caractere exceto newline
^ # o início da linha ou do caractere
$ # o fim da linha ou do caractere
* # nenhum, um ou mais do último caractere
+ # um ou mais do último caractere
? # nenhum ou o último caractere
```

e aqui alguns exemplos de comparações. Recorde que seu uso é sempre entre barras.

```
p.r # p seguido por qualquer caractere seguido por e
# isto irá comparar:
# par
```

```
# para
# por
^f # f no início da linha
^ftp # ftp no início da linha
es$ # es no final da linha
und* # un seguido por nenhum ou vários caracteres d
# isto irá comparar:
# un
# und
# undd
# unddd (etc)
.* # qualquer string sem o newline,
# isto ocorre porque o . compara qualquer caractere
# e o * é o mesmo que nenhum ou mais deste caractere
^$ # uma linha vazia
```

Existem ainda mais opções. Colchetes podem ser usados para comparar qualquer um destes caracteres dentro deles. Entre colchetes, o símbolo - indica entre e o símbolo ^ no início significa não:

```
[qjk] # tanto q ou j ou k
[^qjk] # exceto q, j e k
[a-z] # qualquer um entre a e z, inclusive estes
[^a-z] # exceto letras minúsculas
[a-zA-Z] # qualquer letra
[a-z]+ # qualquer sequência com pelo menos uma ou mais letras minúsculas
```

As demais opções servem mais para referência.

Uma barra vertical **|** representa ou e os parênteses podem ser usados para agrupar caracteres:

```
geléia|creme # tanto geléia ou creme
(ov|dad)os # tanto ovos ou dados
(da)+ # tanto da ou dada ou dadada
```

Aqui estão mais alguns caracteres especiais:

```
\n # um newline
\t # um tab
\w # qualquer caractere alfanumérico (palavra)
# o mesmo que [a-zA-Z0-9_]
\W # exceto caracteres alfanuméricos,
# o mesmo que [^a-zA-Z0-9_]
\d # qualquer dígito
# o mesmo que [0-9]
\D # exceto dígitos, [^0-9]
\s # qualquer caractere em branco: espaço,
# tab, newline, etc
\S # exceto caracteres em branco
\b # uma palavra limitada, fora de []
\B # nenhuma palavra limitada
```

Está claro que caracteres como **$, |, [,), \, /** são casos particulares em expressões regulares. Se você quiser comparar um ou mais destes caracteres, então deve sempre precedê-los de uma barra invertida:

```
\| # barra vertical
\[ # colchetes esquerdo
\) # parênteses direito
\* # asterisco
\^ # circunflexo
\/ # uma barra
\\ # barra invertida
```

Alguns exemplos de ERs

Como foi mencionado antes, o uso de expressões regulares deve ocorrer aos poucos. Aqui estão alguns exemplos. Lembre-se sempre de usá-los entre barras /.../

```
[01] # tanto "0" ou "1"
\/0 # divisão por zero
\/ 0 # divisão por zero com um espaço
\/\s0 # divisão por zero com um espaço em branco
# "/ 0" onde o espaço pode ser um tab, por exemplo
\/ *0 # divisão por zero com nenhum ou alguns espaços
# "/0" ou "/ 0" ou "/ 0" etc
\/\s+0 # divisão por zero com alguns espaços em
branco
\/\s+0\.0* # como o anterior, mas com ponto decimal
# e alguns zeros depois dele, ou não
# "/ 0." ou "/ 0.0" ou "/ 0.000" etc
# observe a barra invertida antes do ponto
```

Exercícios Seu programa anterior contava linhas não-vazias. Altere ele de modo que conte somente linhas com:
a letra x a string para a string para, sendo p maiúsculo ou minúsculo a palavra para ou Para, use **\b** para detectar palavras limitadas com um espaço em branco (newline, tab, etc)
Em cada caso, o programa deve mostrar toda linha, mas somente as específicas devem ser numeradas. Tente usar a variável **$_** para evitar o uso do operador de comparação =~.

Substituição e tradução

Uma vez identificadas as expressões regulares, Perl permite fazer substituições baseadas naquelas comparações.
O modo de fazer isso é usando a função s que é parecida com o modo de substituição usado pelo editor de texto vi do Unix.
Mais uma vez o operador de comparação é usado, e mais uma vez se ele for omitido então a substituição utilizará a variável **$_** em seu lugar.
Para substituir uma ocorrência de londres por Londres na $sentença devemos usar a expressão:

```
$sentenca =~ s/londres/Londres/
```

e para fazer o mesmo com a variável **$_** use:

```
s/londres/Londres/
```

Note que as duas expressões regulares estão usando um total de três barras. O resultado desta expressão é o número de substituições feitas, de modo que pode ser tanto zero (falso) como um (verdadeiro), no caso acima.
Opções Recordando modelos Tradução Exercícios

Opções

Este exemplo somente substitui a primeira ocorrência da string, mas pode ser que tenha mais do que uma string que desejamos substituir. Para fazer uma substituição global, a última barra deve ser seguida por um g:

```
s/londres/Londres/g
```

que obviamente utiliza a variável **$_**. Novamente a expressão retorna o número de substituições realizadas, que é zero ou alguma coisa maior que zero (verdadeiro).

Se desejarmos também substituir ocorrências de lOndres, lonDRES, LoNDrES e assim por diante, então podemos usar:

```
s/[Ll][Oo][Nn][Dd][Rr][Ee][Ss]/Londres/g
```

mas um modo mais fácil é usar a opção i (para ignorar letras maiúsculas ou minúsculas). A expressão:

```
s/londres/Londres/gi
```

fará uma substituição global não importando se foi usado letras maiúsculas ou minúsculas. A opção i também pode ser usada em expressões regulares básicas.

Recordando modelos É frequentemente útil recordar modelos que tem sido comparados de modo que eles possam ser usados novamente. Isto ocorre porque qualquer coisa comparada entre parênteses retorna nas variáveis $1,...,$9. Estas strings também podem ser usadas da mesma forma que nas expressões regulares (ou substituições) utilizando os códigos especiais ER \1,...,\9.
Esse teste:

```
$_ = "Lorde Whopper de Fibbing";
s/([A-Z])/:\1:/g;
print "$_\n";
```

colocará cada letra maiúscula entre : (dois pontos). Neste exemplo, mostrará :L:orde :W:hopper de :F:ibbing. As variáveis $1,...,$9 são somente de leitura, você não pode alterá-las.
Como outro exemplo, o teste:

```
if (/(\b.+\b) \1/)
{
print "Encontrou $1 repetida\n";
}
```

irá identificar qualquer palavra repetida. Cada **\b** representa uma palavra limitada e .+ compara qualquer string não vazia, então **\b.+\b** compara qualquer coisa entre duas palavras limitadas.
O resultado é então armazenado como **\1** para expressões regulares e como **$1** para o resto do programa.
Se **$_** for igual a "teste teste teste2 teste2", a mensagem será mostrada, mas somente com a primeira ocorrência.
O seguinte exemplo troca o primeiro e último caracteres da linha na variável **$_**:

```
s/^(.)(.*)(.)$/\3\2\1/
```

O ^ e o **$** comparam o início e o fim da linha. O código **\1** armazena o primeiro caractere, o **\2** armazena cada string entre os dois e o último caractere é armazenado no código **\3**. Então aquela linha é substituída com a troca entre **\1** e **\3**.
Após uma comparação, você pode usar variáveis especiais somente de leitura **$`** ou **$&** ou **$'** para encontrar o que foi comparado antes, durante e depois da busca. Então:

```
$_ = "Lorde Whopper de Fibbing";
/pp/;
```

resultará em verdadeiro nas seguintes declarações (recorde que eq é um teste usado em strings):

```
$` eq "Lorde Who";
$& eq "pp";
$' eq "er de Fibbing";
```

Finalmente, sobre o tópico Recordando modelos, o importante é saber que dentro de barras de uma comparação ou substituição, as variáveis são interpoladas. Então:

```
$busca = "par";
s/$busca/xxx/g;
```

irá substituir cada ocorrência com xxx. Se você quiser substituir cada ocorrência de para então você não pode usar **s/$buscaa/xxx/g** porque o a será interpolado com a variável $busca. Ao invés disso, você deve colocar o nome da variável entre chaves, de modo que o código seja:

```
$busca = "par";
s/${busca}a/xxx/g;
```

Tradução A função tr permite a tradução caractere-a-caractere. A seguinte expressão substitui cada a com e, cada b com d, cada c com f na variável $sentença. A expressão retorna o número de substituições efetuadas:

```
$sentenca =~ tr/abc/edf/
```

A maioria dos códigos especiais ER não se aplica à função tr. Por exemplo, a próxima declaração conta o número de asteriscos na variável $sentença e armazena na variável $contagem:

```
$contagem = ($sentenca =~ tr/*/*/);
```

Contudo, o traço permanece como seu uso entre. Esta declaração converte toda $_ para letra maiúscula:

```
tr/a-z/A-Z/;
```

Exercícios

Seu último programa contava linhas de um arquivo que continham uma certa string.
Modifique ele de modo que conte linhas com letras duplas (ou qualquer outro caractere duplo).
Modifique novamente para que estas letras duplas apareçam entre parênteses. Por exemplo, seu programa deve produzir algo como:

```
023 E(ss)es pioneiros conduziram muitas se(ss)ões
```

Tente fazer com que todos os pares de letras sejam colocados entre parênteses, e não apenas o primeiro par de cada linha.
Para um programa mais interessante, você deve tentar o seguinte. Suponha que seu programa se chama contalinhas. Então você deve executá-lo com:

```
perl contalinhas
```

Contudo, se você chamá-lo com vários argumentos, como em:

```
perl contalinhas primeiro segundo etc
```

então estes argumentos serão armazenados em uma matriz **@ARGV**.
No exemplo acima, nós teremos **$ARGV[0]** igual a primeiro, **$ARGV[1]** igual a segundo e **$ARGV[2]** igual a etc.
Modifique seu programa de modo que ele aceite uma string como argumento e conte somente linhas com essa string. Ele deve também colocar ocorrências dela entre parênteses. Então:

```
perl contalinhas par
```

deverá retornar algo como:

```
019 Mas (par)a os grandes pioneiros, suas (par)tes
eram...
```

Split Uma função muito útil no Perl é a split, que separa uma string e coloca em uma matriz. A função usa expressões regulares e também funciona com a variável especial $_.
A função split é usada desta forma:

```
$info = "Caine:Michael:Ator:Doce Liberdade";
@pessoal = split(/:/, $info);
```

que tem o mesmo efeito que:

```
@pessoal = ("Caine", "Michael", "Ator", "Doce Liberdade");
```

Se a informação estiver armazenada na variável $_ então poderemos usar somente:

```
@pessoal = split(/:/);
```

Se os campos são divididos por vários dois pontos, então podemos usar os códigos ER para certos problemas. O código: `$_ = "Caine::Michael:::Ator";`

```
@pessoal = split(/:+/);
```

é o mesmo que:

```
@pessoal = ("Caine", "Michael", "Ator");
```

mas se não for usado, então:

```
$_ = "Caine::Michael:::Ator";
@pessoal = split(/:/);
```
será o mesmo que:
```
@pessoal = ("Caine", "", "Michael", "", "", "Ator");
```
Uma palavra pode ser separada em caracteres, uma sentença separada em palavras e um parágrafo em sentenças:
```
@caracteres = split(//, $palavra);
@palavras = split(/ /, $sentenca);
@sentencas = split(/\./, $paragrafo);
```
No primeiro caso, a string nula é comparada entre cada caractere, e isso ocorre porque @caracteres é uma matriz de caracteres, i.é, uma matriz de strings com comprimento 1.
Exercícios

Exercícios Uma ferramenta útil no processamento de nossa língua natural é a concordância. Este permite que uma string específica seja mostrada em seu contexto imediato, em qualquer lugar que apareça no texto.
Por exemplo, um programa de concordância identificando a string destino par talvez possa produzir alguma das seguintes saídas. Note como as ocorrências da string destino se alinham verticalmente:
o primeiro par de ligações ocorreu
sendo que para entender os proble
no segundo par existe a possibilid
s o último par deve conter a final
Neste exercício, você deve fazer esse programa. Aqui estão algumas dicas:
coloque o arquivo inteiro em uma matriz (geralmente isso não é útil porque o arquivo pode ser extremamente grande, mas não devemos nos preocupar com isso agora). Cada item na matriz será uma linha do arquivo. quando a função chop é usada em uma matriz, ela elimina o último caractere de cada item da matriz. recorde que você pode juntar a matriz inteira com uma declaração como **$texto = "@linhas"**; use a string destino como delimitador para separar o texto (i.é, use a string no lugar dos dois pontos usados nos exemplos anteriores). Você deve então ter uma matriz de todas as strings entre cada string destino. para cada elemento da matriz, mostre ele, a string destino, e então mostre o próximo elemento da matriz. recorde que o último elemento de uma matriz @comida tem índice $#comida.
Este deveria ser um bom programa, mas a string destino não se alinhará verticalmente. Para alinhar essas strings, você precisará da função substr. Aqui estão três exemplos de seu uso:
```
substr("Era uma vez a...", 3, 4); # retorna " uma"
substr("Era uma vez a...", 7); # retorna " vez a..."
substr("Era uma vez a...", -6, 5); # retorna "z a.."
```
O primeiro exemplo retorna uma substring de comprimento 4 começando a partir da posição 3. Recorde que o primeiro caractere de uma string tem índice zero. No segundo exemplo, se faltar o comprimento, toda a substring a partir da posição 7 é mostrada. O terceiro exemplo mostra que você também pode contar a partir do final, usando um índice negativo. Ele retorna a substring que começa a partir do sexto caractere contado da direita para a esquerda com comprimento de 5 caracteres.
Se você usar um índice negativo que estende além do começo da string, então o Perl retornará nada ou uma mensagem de erro. Para evitar que isso ocorra, você pode aumentar o tamanho da string usando o operador x, como mencionado antes. Por exemplo, a expressão **(" "x30)** produz 30 espaços.

Matrizes associativas Uma lista comum de matriz permite acessar seus elementos através de números. O primeiro elemento da matriz @comida é $comida[0]. O segundo é $comida[1], assim por diante, sendo que o último elemento é $comida[$#comida].
Mas o Perl também permite criar matrizes que são acessadas por uma string. Estas são chamadas de matrizes associativas.
Para definir uma matriz associativa, usamos a notação usual com parênteses, mas a matriz agora é prefixada por um sinal %. Suponha que nós queremos criar uma matriz de pessoas com suas idades. Ela pode ser algo como:

```
%idades = ("Michael Caine", 39,
"Den", 34,
"Angie", 27,
"Willy", "21 anos de cachorro",
"A rainha mãe", 108);
```

Agora podemos encontrar a idade de cada pessoa com as seguintes expressões:

```
$idade{"Michael Caine"}; # Retorna 39
$idade{"Den"}; # Retorna 34
$idade{"Angie"}; # Retorna 27
$idade{"Willy"}; # Retorna "21 anos de cachorro"
$idade{"A rainha mãe"}; # Retorna 108
```

Note que como na matriz comum, cada sinal % foi alterado para um $ para acessar um elemento individual porque cada elemento é escalar. Mas, de forma diferente da lista comum, onde o índice é colocado entre parênteses, a idéia está em facilitar o acesso através da associação de cada par de elementos.

Uma matriz associativa pode ser convertida em uma comum apenas atribuindo ela a uma variável @, e vice-versa. Note porém que a conversão não coloca os elementos pares em ordem, como veremos adiante;

```
@info = %idades; # @info é uma lista comum. Ela tem agora 10 elementos
$info[5]; # retorna o valor 27
%maisidades = @info; # %maisidades tem agora 5 pares de elementos
```

Operadores Variavéis de Ambiente

Operadores Matrizes associativas não ordenam seus elementos (elas são como grandes tabelas confusas) mas é possível acessar todos seus elementos usando funções de chaves e funções de valores:

```
foreach $pessoa (keys %idades)
{
print "Eu sei a idade de $pessoa\n";
}
foreach $idade (values %idades)
{
print "Alguém tem $idade\n";
}
```

Quando keys é chamada, ela retorna uma lista de índices da matriz associativa. Quando values é chamada, ela retorna uma lista dos valores da matriz. Essas funções retornam suas listas na mesma ordem, mas esta ordem não tem nada a ver com a ordem em que os elementos foram inseridos.

Quando keys e values são chamadas em um contexto escalar, elas retornam o número de pares na matriz associativa.

Existe também um função que retorna uma lista de dois elementos de uma chave e seu valor. Cada vez que é chamada, ela retorna outro par de elementos:

```
while (($pessoa, $idade) = each(%idades))
{
print "$pessoa tem $idade\n";
}
```

Variavéis de Ambiente Quando você roda um programa em Perl, ou qualquer script no Unix, certas variáveis de ambientes são setadas. Algumas podem ser como **$ENV{'USER'}** que contém o username (nome do usuário logado).

Quando você roda um script CGI na WWW, existem variáveis de ambiente que possuem outras informações úteis. Todas essas variáveis e seus valores são armazenados na matriz associativa **%ENV** em que as chaves são os nomes das variáveis. O seguinte exemplo retorna todas as suas variáveis de ambiente:

```
while (($chave, $valor) = each(%ENV))
{
print "$chave é igual a $valor\n";
}
```

Subrotinas Como qualquer linguagem estruturada, Perl permite ao usuário definir suas próprias funções, chamadas subrotinas. Elas podem ser colocadas em qualquer lugar do programa, mas o ideal é colocá-las no início, ou tudo no final. Uma subrotina tem sempre o formato:

```
sub subrotina
{
print "Esta é uma rotina muito simples.\n";
print "Ela sempre faz a mesma coisa...\n";
}
```

As seguintes declarações servem para chamar a subrotina. Note que ela é invocada com um caractere & na frente do nome:

```
&subrotina; # chama a subrotina
&subrotina($_); # chama ela com um parâmetro
&subrotina(1+2, $_); # chama com dois parâmetros
```

Parâmetros Retornado Valores Variáveis Locais

Parâmetros

Nos casos acima, os parâmetros são aceitos porém ignorados. Quando a subrotina é chamada, os parâmetros são passados como uma lista na variável especial @_. Esta variável não tem nada a ver com a variável escalar $_. A seguinte subrotina simplesmente mostra a lista de argumentos passados:

```
sub argumentos
{
print "@_\n";
}
```

Alguns exemplos de seu uso:

```
&argumentos("primeiro", "segundo"); # mostra primeiro
segundo

&argumentos("maçãs", "e", "uvas"); # mostra maçãs e
uvas
```

Como em qualquer outra matriz comum, os elementos individuais de @_ podem ser acessados com a notação entre colchetes:

```
sub primeiros
{
print "Primeiro argumento: $_[0]\n";
print "Segundo argumento: $_[1]\n";
}
```

Novamente é importante recordar que as variáveis escalares indexadas **$_[0], $_[1]**, etc não têm nada a ver com a variável $_, que pode ser usado normalmente sem causar problemas.

Retornando valores

O resultado de uma subrotina é sempre a última coisa a se realizar. Esta subrotina retorna o maior valor:

```
sub maior
{
if ($_[0] > $_[1])
{
$_[0];
}
else
{
$_[1];
}
}
```

Neste exemplo, $maior será igual a 37:

```
$maiorvalor = &maior(37, 24);
```

A subrotina &primeiros também retorna um valor, que neste é caso 1.
Isto ocorre porque a última coisa que aconteceu foi a execução da
função print, e quando essa declaração funciona corretamente, ela
resulta em 1.

Variáveis locais A variável @_ é local somente para a corrente subrotina, e assim são as demais, como **$_[0], $_[1]**, etc. Outras variáveis também podem ser locais, e isto é útil se quisermos começar alterando os parâmetros passados. A seguinte subrotina verifica se uma string está dentro de outra, comparando sem os espaços entre as palavras:

```
sub dentro
{
local($a, $b); # cria variáveis locais
($a, $b) = ($_[0], $_[1]); # atribui valores
$a =~ s/ //g; # retira espaços das
$b =~ s/ //g; # variáveis locais
($a =~ /$b/ || $b =~ /$a/); # $b está dentro de $a,
# ou $b contém $a?
}
```

A seguinte expressão retorna verdadeiro:

```
&dentro("mamão", "lima mão");
```

De fato, você também pode eliminar redundâncias substituindo as duas primeiras declarações do bloco pela linha:

```
local($a, $b) = ($_[0], $_[1]);
```