Formação Profissional
WEB DESIGNER

IBPI NET
The Internet School

Introdução à linguagem perl

Aborda Perl 4 e Perl 5

Material extra: CWD-2
Atualizado em 03/2000

Helder da Rocha

WebDesigner / IBPINET

Criação de Web Sites II

Material de referência
Introdução à Linguagem Perl

Conteúdo

Introdução à linguagem Perl

ESTE APÊNDICE destina-se às pessoas que têm alguma familiaridade com linguagens de programação de computadores. A finalidade deste material é servir como referência para quem quiser se aprofundar um pouco mais nessa linguagem com o objetivo de desenvolver (ou alterar) programas CGI. O texto contém uma explicação da estrutura, sintaxe e recursos da linguagem seguida de exemplos com trechos de código. Apesar de abordar assuntos que fogem ao escopo do curso de Web Designer, esta introdução não é exaustiva e é insuficiente para quem tiver como objetivo conhecer Perl a fundo. Para saber utilizar recursos como controle de processos, ponteiros e construção de soquetes de rede (TCP/IP) você deve procurar documentação mais detalhada como referências listadas no final.

Tópicos abordados neste capítulo

- Introdução à linguagem *Perl*
- Sintaxe básica e estrutura
- Entrada e saída de dados via dispositivos padrões e arquivos
- Expressões regulares do *Unix* disponíveis em *Perl*
- Criação e utilização de funções
- Transformação de *strings*, formatos e outros tópicos avançados
- Uso elementar de módulos do *Perl* 5

Objetivos

No final deste capítulo você deverá ser capaz de:

- Conhecer a sintaxe básica da linguagem *Perl* e Utilizar os recursos disponíveis na linguagem para desenvolver programas utilitários utilizáveis em *Unix* e *Windows*.
- Analisar um programa simples em *Perl* (como um *script* CGI), identificar suas partes, entendê-lo de uma forma geral (saber o que ele faz) e saber alterá-lo (programas em *Perl 4* ou usando módulos apresentados aqui).
- Saber o que são expressões regulares e como usá-las.
- Incluir módulos e bibliotecas em um programa e conhecer a sintaxe básica utilizada para construir objetos e chamar seus métodos. Para usar os módulos você deverá consultar a documentação de cada um.

1. Sintaxe básica

De acordo com seu criador, Larry Wall, *Perl* é abreviação de *Practical Extraction and Report Language* (Linguagem para a Extração Prática e Relatórios) e também de *Pathological Ecletic Rubbish Lister* (Listador de Lixo Patologicamente Eclético). Foi criado para auxiliar o usuário e administrador *Unix* a realizar tarefas que são muito complicadas para o *Shell* (linguagem de propósito geral para automação de tarefas do *Unix*) mas ao mesmo tempo de vida muito curta ou complicada demais para escrever em *C* ou outra linguagem do *Unix*. Por causa dos seus recursos de formatação de texto e recursos disponíveis para o desenvolvimento de programas pequenos, porém sofisticados, *Perl* se tornou a linguagem preferencial dos desenvolvedores CGI.

Perl é uma linguagem interpretada. Para executar um programa em *Perl*, ele deve ser armazenado em um arquivo com extensão `.pl` e executado através da invocação do interpretador, da forma:

```
perl seuprog.pl
```

Em sistemas *Unix*, qualquer *script* pode ser transformado em um programa executável. Para isto, é preciso que o programa informe, na primeira linha do código, onde está o interpretador. O clássico e insuportável programa *Hello World* pode ser escrito em *Perl* da seguinte maneira:

```
#!/usr/local/bin/perl
print "Hello, world!\n";
```

A primeira linha é um comentário para a linguagem *Perl*. Em sistemas *Unix*, diz que este é um programa em *Perl* e informa o endereço do interpretador, o que permite que o programa seja executado diretamente sem precisar chamá-lo, na linha de comando, ou seja, pode-se fazer simplesmente:

```
./seuprog.pl
```

para executar o programa. É necessário que o bit executável do arquivo seja ligado para que isto funcione

```
chmod a+x seuprog.pl
```

Em sistemas *Windows*, o comentário da primeira linha geralmente é ignorado. Mas, em programas CGI, alguns servidores Web como o *Apache* utilizam a primeira linha no *Windows* para localizar o interpretador, por exemplo:

```
#c:\perl\bin\perl.exe
print "Hello, world!\n";
```

A maior parte dos servidores Web, porém, utiliza-se de outra forma de associação entre o interpretador e o programa.

No exemplo acima, a segunda linha é o código executável do programa. A instrução `print` imprime o seu argumento na *saída padrão*. No final do *string* que é argumento de `print`, há um código de escape especial: "`\n`". Este caractere provoca uma quebra de linha. Observe que a linha termina em ";". Isto ocorre em *todas* as instruções simples do *Perl*.

Para rodar o programa, basta salvá-lo (como "`hello.pl`", por exemplo) e rodá-lo:

```
perl hello.pl
```

ou diretamente,

```
./hello.pl
```

se o programa estiver em ambiente *Unix* com bit executável ligado e endereço do interpretador informado corretamente. O resultado será

```
Hello, world!
```

1.1. Instruções

A sintaxe das instruções em *Perl* é semelhante à de *C* (ou *Java*). Toda instrução simples termina com um "`;`" (ponto-e-vírgula). Instruções compostas podem ser agrupadas entre chaves (funções, estruturas `for`, `while`, etc.). O espaço em branco (tabulações, quebras de linha, espaços) é ignorado, exceto dentro de *strings*.

Para programar em *Perl*, pode-se utilizar o alfabeto ISO-Latin-1 (8 bits). Os identificadores, porém, devem limitar-se ao alfabeto ASCII.

Comentários valem até o fim de uma linha e são precedidos do caractere #.

1.2. Literais e identificadores

Literais são representações dos valores escalares utilizados para representar números ou *strings*. Pode-se transformar dados escalares (como somar, concatenar) e armazená-los em variáveis representadas por identificadores.

Todos os valores numéricos têm, internamente, o mesmo formato em *Perl*: ponto-flutuante de dupla precisão, embora possam ser representados de várias maneiras. Os literais numéricos que representam números de ponto-flutuante podem conter ponto decimal, sinal e a letra "`e`", indicando expoente (`12.3e-34`). Os que representam inteiros são por *default*, decimais. Podem representar valores octais se precedidos por "`0`" ou hexadecimais, se precedidos por "`0x`".

Strings são seqüências de caracteres. Cada caractere tem 8 bits (*Perl* 4). O menor *string* é o *string* vazio. O maior, ocupa toda a memória disponível. Como você pode manipular com *strings* de 8 bits, valores binários podem ser manipulados em *Perl* se forem tratados como *strings*.

Há três representações diferentes para *strings* em *Perl*. *Strings* entre apóstrofes (aspas simples) é uma seqüência de caracteres armazenada entre apóstrofes. Não há caracteres ilegais. Até o caractere de quebra de linha (se a linha quebrar dentro do *string*) é considerado parte dele. Tais *strings* podem, portanto, ser usados para até com dados binários.

```
$texto_longo = "
<HTML>
<HEAD><TITLE>A Divina Comédia - Inferno: Canto 1</TITLE></HEAD>
<BODY><TABLE WIDTH=450><TR><TD>
<H1>Canto I</H1>
<P>Quando eu me encontrava na metade do caminho de nossa vida, me vi
perdido em uma selva escura, e a minha vida não mais seguia o cami-
nho certo.
(...)
```

```
</TD></TR></TABLE></BODY></HTML>
"; # este string não é neutro (seria se estivesse entre ' e ')
```

Alguns caracteres, quando aparecem dentro de strings, podem determinar o seu final ou terem uma interpretação diferente da desejada. Esses caracteres, para que possam aparecer na sua forma original, devem ser precedidos de uma contr-barra. As únicas exceções dentro de um *string* de apóstrofes são o apóstrofe, se ocorrer dentro do *string*, e a contra barra. Para produzir uma contra-barra, deve-se usar duas contra-barras. Para produzir um apóstrofe, deve-se precedê-lo por uma contra-barra.

Strings entre aspas (duplas) não são neutros. Variáveis escalares neles contidos, cifrões, arrobas, porcentagens podem ter uma interpretação diferente se não forem precedidos de contra-barra. Além disso, há várias seqüências de escape formadas por certos caracteres especiais precedidos pela contra-barra para produzir efeitos especiais. Veja na tabela abaixo.

SEQÜÊNCIA	VALOR DO CARACTERE
`\b`	Retrocesso (backspace)
`\t`	Tabulação
`\n`	Nova Linha (new line)
`\f`	Alimentação de Formulário (form feed)
`\r`	Retorno de Carro (carriage return)
`\v`	Tabulação vertical
`\a`	Bell
`\e`	Escape
`\cX`	Qualquer caractere de controle (aqui, `^X`)
`\l`	Próximo caractere em caixa baixa
`\L`	Próximos caracteres em caixa baixa até o `\E`
`\u`	Próximo caractere em caixa alta
`\U`	Próximos caracteres em caixa alta até o `\E`
`\E`	Terminador de `\U` ou `\L`
`\"`	Aspas
`\'`	Aspa (apóstrofe)
`\\`	Contra Barra
`\`*nnn*	O caractere correspondente ao valor octal *nnn*, onde *nnn* é um valor entre 000 e 037.
`\x`*nn*	O caractere de 8-bits *nn*, onde *nn* é de um a dois dígitos hexadecimais.

A terceira categoria de *strings* em *Perl* não representa cadeias de caracteres mas, comandos do sistema. Comandos entre crases `` `dir c:` `` são executados e seu resultado é retornado. Tais escapes tornam o programa dependente de plataforma e devem ser evitados caso se pretenda rodar o programa em mais de uma plataforma.

Comandos do sistema também podem ser executados através do operador `system`:

```
system "dir c:";
```

1.3. Operadores

Perl oferece operadores diferentes para *strings* e números. Os operadores aritméticos são os mesmos usados em *Java*, *C*, *C++* e linguagens semelhantes. As regras de precedência das operações básicas também são as mesmas, mas *Perl* tem regras de precedência próprias para diversos outros operadores.

OPERADOR	FUNÇÃO	OPERADOR	FUNÇÃO
+	Adição	~	Complemento
-	Subtração	=	Atribuição simples
*	Multiplicação	+=	Atribuição com soma
/	Divisão	-=	Atribuição com subtração
%	Resto	*=	Atribuição com multiplicação
**	Exponenciação	/=	Atribuição com divisão
++	Incremento	%=	Atribuição com resto
--	Decremento	&=	Atribuição com AND
!	NÃO lógico	|=	Atribuição com OR
&&	E lógico	^=	Atribuição com XOR
||	OU lógico	?:	Operador ternário *if/then/else*
&	AND	x	Repetição de *string*
^	XOR	.	Concatenação de *string*
|	OR	-*op*	Teste de arquivo (*op:* e, d, ...)
>>	Desloc. de bits à direita	=~	Expr. regular combina
<<	Desloc. de bits à esquerda	=!	Expr. regular não combina

A tabela acima não inclui os operadores usados para realizar comparações booleanas. Para comparar números ou *strings*, os operadores não são os mesmos. A tabela abaixo mostra as diferenças:

COMPARAÇÃO	STRING	NÚMERO
Maior	gt	>
Menor	lt	<
Maior ou igual	ge	>=
Menor ou igual	le	<=
Igual	eq	==
Diferente	ne	!=

Por exemplo:

```
if ($metodo eq "POST") {
    if ($numero >= 25) {
        ....
```

Strings podem ser concatenados usando o operador "." (ponto). Quaisquer números que estiverem envolvidos serão convertidos em *strings* antes da concatenação. Se *strings*, que tiverem partes numéricas, forem usados em operações aritméticas, eles também serão convertidos em números (se for possível, ou seja, se alguma parte desses *strings* representar um número). As partes não numéricas serão descartadas.

O operador x é usado para repetir *strings*. É um operador não comutativo (como o operador de concatenação) e transformará em *strings* qualquer coisa que estiver à sua esquerda. À direita, este operador recebe um número, que informa quantas vezes um determinado *string* deve ser repetido:

```
"Au" x 9; # imprime AuAuAuAuAuAuAuAuAu
"dinheiro" x 3; # imprime dinheirodinheirodinheiro
```

1.4. Variáveis escalares

Variáveis escalares são representadas por um identificador consistindo de um cifrão ($), seguido por uma letra, e depois, talvez, por números e/ou letras. Cria-se uma variável, atribuindo-lhe um valor escalar (número ou *string*). Não é necessário declarar variáveis, basta usá-las. As variáveis não têm tipo. Podem armazenar valores de tipos diferentes a qualquer momento.

```
$a = 13;
$b = $a + 26;
$b++;    # operação de incremento
```

A operação mais comum sobre variáveis escalares é a atribuição. O operador de atribuição em *Perl* é o sinal de igualdade (=). Uma atribuição pode ser combinada com um operador e formar uma atribuição binária:

```
$a = $a + 1; # esta operação, faz a mesma coisa que...
$a += 1; #atribuição binária (atribuição com soma) ... e
$a++;    # incremento unário
```

O operador `chop` pode ser usado apenas com variáveis escalares para arrancar o último caractere de um *string*. Isto é útil quando se lê dados da entrada padrão. A operação atua sobre a variável e retorna o caractere arrancado:

```
$gelado = "abcd";
chop $gelado;       # $gelado agora contém "abc"
$y = chop $gelado;  # $gelado contém "ab" e $y contém "c"
```

Variáveis que ocorrem dentro de *strings* de aspas duplas não são ignoradas. Dentro de tais *strings*, portanto, se houver necessidade de se imprimir um cifrão, é preciso precedê-lo por uma contra-barra. Outra solução é colocar o cifrão dentro de um *string* neutro (de apóstrofes):

```
$a = "texto";
$b = "Eis uma linha de $a"; # produz "Eis uma linha de texto"
$c1 = "Eis uma linha de \$a";  # produz "Eis uma linha de $a"
$c2 = "Eis uma linha de ".'$a';  # produz "Eis uma linha de $a"
```

Variáveis usadas antes de terem um valor atribuído a elas possuem o valor `undef`. Esse valor se traduz como false, zero ou *string* nulo dependendo do contexto onde é utilizado.

Strings de múltiplas linhas podem ser atribuídos a variáveis de duas formas. A primeira, como já vimos, usando atribuição simples:

```
$texto = "
Texto de mais de uma linha.
Segunda linha.
Terceira.
";
```

Uma outra forma é usar o operador "<<", que permite abrir um espaço para um longo *string*, que só termina quando um delimitador escolhido for encontrado sozinho em uma linha. O efeito do bloco abaixo é o mesmo que o anterior:

```
$texto = <<FIM;
Texto de mais de uma linha.
Segunda linha.
Terceira.
FIM
```

Não pode haver qualquer outra coisa na linha que possui o delimitador ou o programa não o encontrará (nem ponto-e-virgula). Deve haver uma quebra de linha imediatamente após o nome e não deve haver qualquer espaço antes.

1.5. Vetores indexados

Um vetor indexado é uma lista ordenada de valores escalares. Cada elemento é uma variável escalar separada com um valor independente, que pode ser de qualquer tipo. Um vetor pode ter qualquer quantidade de elementos. Não é preciso declará-los. Podem crescer ou diminuir à vontade, conforme necessário.

Vetores são representados por literais que consistem de um par de parênteses (o vetor vazio), um par de parênteses contendo um escalar (vetor unitário) ou um par de parênteses contendo vários escalares, separados por vírgulas. Os valores podem ser constantes, variáveis, expressões ou até outros vetores (cujos elementos são incorporados e fazem crescer o vetor):

```
(1, 2, 3, 4, 5)
($a, $b, $c) = (1, 2, 3) # atribui 1 a $a, 2 a $b, etc.
($a + 15, "anta", -17, $x)
(1..4, 10..14) # mesmo que (1, 2, 3, 4, 10, 11, 12, 13, 14)
```

1.6. Variáveis de vetores

Uma variável de vetor contém um único vetor (com sua coleção de valores escalares). Os identificadores são idênticos aos usados com escalares, exceto que, estes, começam com `@` em vez de `$`. Por exemplo:

```
@vetor = ($a, $b, $c, $d);
($x, $y) = @vetor; # $x contém $a e $y contém $b
@maior = (1, @vetor, 2, 3) #  mesmo que (1, $a, $b, $c, $d, 2, 3)
@dois = (1, 2);
($u, $d, %t) = @dois; # contém (1, 2, undef)
```

Uma variável de vetor que ainda não tenha sido inicializada com uma atribuição contém o vetor vazio (sem elementos) representado pelo par de parênteses `()`. Observe que os elementos dos vetores inseridos têm o mesmo nível dos elementos originais. Se o vetor só contém referências escalares (como o segundo e últimos exemplos acima), pode ser usado do lado esquerdo da atribuição. Se um vetor for atribuído a um escalar, este guardará o seu comprimento (número de elementos):

```
$numElem = (1, 20, 46, 89) # $numElem contém 4
```

Os elementos dos vetores são acessíveis individualmente através de seu índice usando colchetes. A referência é o próprio identificador do vetor na sintaxe de escalar (precedido de `$`) e seguido de um par de colchetes `[` e `]` contendo um número (ou expressão que resulta em número) correspondente ao elemento. A contagem dos elementos começa em zero:

```
@planetas = ("mercúrio","vênus","terra","marte");
$x = $planetas[2];         # $x contém "terra"
$planetas[1] = "Ishtar";   # troca vênus por Ishtar

$planetas[5] = "Saturno"; # @planetas agora contém ("mercúrio",
                          # "Ishtar","terra","marte",undef,"saturno")
$a = 2;
$b = $planetas[$a - 1]     # contém "Ishtar"
```

Observe que a variável que tem acesso ao elemento do vetor é escalar. Pode-se usar um variável de vetor, porém, para obter apenas uma parte do vetor original:

```
@dois = @planetas[1, 2]; # @dois contém ($planetas[1], $planetas[2])
@tres = (5,10,15,20,25,30,35,40) [4,5,6]; # @tres contém (25, 30, 35)
```

1.7. Operadores de vetores

O vetor pode ser manipulado como uma pilha usando os operadores `push` e `pop` para acrescentar ou remover os últimos elementos de uma lista:

```
push (@planetas, $novoAstro) ; # empurra $novoAstro no fim de @planetas
$ultimo = pop (@planetas);     # remove o último elemento
push (@x, 1, 2, 3);            # mesmo que (@x, 1, 2, 3)
```

Os operadores `push` e `pop` operam no fim do vetor. Para operar no início, pode-se usar `shift` e `unshift`. O operador `reverse` retorna os elementos do vetor na ordem inversa e `sort` os ordena de acordo com a ordem ascendente dos caracteres ASCII:

```
@z = (1, 2, 3, 10, 20, 30, 100, 200, 300);
@a = reverse @z;   # @a contém (300,200,100,30,20,10,3,2,1)
@s = sort @z;      # @s contém (1, 10, 100, 2, 20, 200, 3, 30, 300)
```

Para que `sort` faça a ordenação de acordo com uma outra regra (numérica, por exemplo), é preciso passar dois argumentos para o operador. O primeiro deve ser uma subrotina (ou corpo da subrotina) que retorne `-1`, `0` ou `1` se o primeiro elemento é menor, igual ou maior, respectivamente, que o segundo.

```
@s = sort &ord_numerica @z;  # &ord_numerica é uma chamada de
                             # subrotina definida localmente
```

Para ordenar pela ordem numérica, pode-se usar o operador auxiliar <=>, no corpo de uma subrotina definida após o `sort` da forma:

```
@s = sort {$a <=> $b} @z;  # (1, 2, 3, 10, 20, 30, 100, 200, 300)
```

Se o operador `chop` for aplicado em um vetor, arrancará o último caractere de cada um de seus elementos.

Expressões pode ser calculadas no contexto escalar ou no contexto de vetor. Por exemplo, se `@x` for impresso, pode-se desejar imprimir seu conteúdo (contexto de vetor) ou seu número de elementos (contexto escalar):

```
@x = ("uga","uga","uh");
print ("Ele disse", @x,".");    # imprime "Ele disse ugaugauh."
print ("Ele disse", "".@x,"."); # imprime "Ele disse 3."
```

No exemplo acima, `@x` foi concatenada com o *string* nulo `""`, o que a transformou em escalar (contém agora o número de elementos do vetor).

Se uma variável de vetor aparecer dentro de uma *string*, será interpretada (assim como ocorre com os escalares). É preciso, portanto, preceder os `@`'s e possíveis `[]`'s com contra-barras (ou colocá-los em *strings* neutros) para que não sejam interpretados como vetores ou elementos de vetor, caso esta seja a intenção:

```
@argos = ("pa","ca"); $argos = "jacaré";
$x = "joao@argos.net";          # imprime "joaopaca.net"
$x = "joao\@argos.net";         # imprime "joao@argos.net"
$y = "A variável é $argos[1]";  #imprime "A variável é ca"
$y = "A variável é $argos\[1]"; #imprime "A variável é jacaré[1]"
```

1.8. Estruturas de controle

As estruturas de controle em *Perl* operam sobre blocos de declarações. Um bloco de declarações é uma instrução composta por várias instruções simples, contidas entre chaves. Um bloco é aceito no lugar de qualquer instrução simples e as suas variáveis (mesmo as criadas dentro do bloco) são globais, a não ser que sejam declaradas locais.

As estruturas de *Perl* são `if-elsif-else`, `unless`, `while`, `until`, `for` e `foreach`. A expressão `if` opera sobre um bloco (que a segue) somente se uma determinada condição (expressa, entre parênteses, após o `if`) for verdadeira. Após o bloco `if` podem existir zero ou mais blocos `elsif`, com outras condições, e, no final, um bloco `else`, complementando os resultados possíveis:

```
if ($x > 10) {
   $b = "é maior";
} elsif ($x < 0) {
   $b = "é menor";
} else {
   $b = "está OK";
}
```

Os valores `undef`, `""` e zero são considerados valores falsos. *Strings* contendo qualquer coisa são considerados verdadeiros, assim como qualquer número diferente de zero.

A expressão `unless` ("a não ser que") é o contrário do `if`. Pode ser usada quando se deseja apenas a parte `else` do `if` (mas ela também pode ter cláusula else):

```
unless ($salario <= 50000) {
 # faça algo somente se $salario não for <= 50000
} else {
 # caso contrário... se salario for <= 50000
}
```

A expressão `while` ("enquanto") permite que se implemente repetições. A expressão `until` ("até que") é o seu oposto. O par `do...while` ("faça ... enquanto") permite que antes do teste, o bloco de instruções seja executado pelo menos uma vez:

```
while($dias < 31) {
   $dias++;
}

until ($dias >= 31) {
   $dias++;
}

do {
   $dias++;
} while($dias < 30);
```

Para repetições em que se sabe previamente o número de vezes em que a operação irá ocorrer, pose-se usar o `for`, que tem a mesma sintaxe que em *C* ou *Java*:

```
for ($i = 0; $i < 100; $i = $i + 10) {
    print "$i carneirinhos... \n";
}
```

A iteração `foreach` permite navegar por uma lista de valores e manipulá-los um por vez como escalar. É ideal para manipular vetores.

```
foreach $galinha (@galinheiro) {
    print $galinha + "\n";
}
```

1.9. Outros operadores de controle de fluxo

Os operadores `last`, `next` e `redo` são usados dentro de uma estrutura de repetição para oferecer meios alternativos de controle de fluxo (além do controle baseado em uma condição verdadeira). O operador `last` serve para forçar a saída de uma repetição (`while`, `do-while` ou `for`). Com `next`, é possível pular partes da estrutura e `redo` permite que se inicie uma nova repetição (deixando de fora trechos que seguem a instrução. Veja os exemplos abaixo:

```
while ($tem_emprego) {
    &bate_ponto;
    if ($demissao) {
        &recebe_indenizacao;
        last;  # cai fora do loop
    }
    &trabalha_dia_inteiro;
    $dias++;
}
# last continua aqui... (não volta mais para o loop)

while ($tem_emprego) {
    &bate_ponto;
    if ($dia_de_folga) {
        &recebe_aviso;
        next;  # pula o resto do loop
    }
    &trabalha_dia_inteiro; # pula!
    $dias++;               # pula!
    # next continua aqui... (segue para fazer o teste do while)
}

while ($tem_emprego) {
    # redo continua aqui... (e pula o teste do while!)
    &bate_ponto;
    if ($ponto_errado) {
        redo;  # pula o resto do loop e o teste
    }
    &trabalha_dia_inteiro;
```

```
    $dias++;
}
```

Os operadores `last`, `next` e `redo` permitem sair na estrutura mais interna. Se for necessário o desvio para um outro bloco (mais externo), é preciso rotulá-lo. O rótulo consiste de um nome, seguido por "`:`" (dois-pontos) que precede o `for` ou `while`.

```
EXT: for ($i = 0; $i < 10; $i++) {
   INT: for ($j = 0; $j < 10; $j++) {
      if ($i == $j) {
          last;  # cai fora do loop atual. Mesmo que last INT;
      } elsif ($i < $j) {
          next EXT; # pula esta rodada do loop externo.
      } else {
           print "($i, $j) "; # imprime (sem quebrar a linha)
      }
   }
   print "\n";  # quebra a linha
}
```

O resultado do programa acima está listado abaixo:

```
(1, 0)
(2, 0) (2, 1)
(3, 0) (3, 1) (3, 2)
(4, 0) (4, 1) (4, 2) (4, 3)
(5, 0) (5, 1) (5, 2) (5, 3) (5, 4)
(6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5)
(7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6)
(8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7)
(9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8)
```

Além das estruturas `if` e `unless`, é possível usar operadores *booleanos* para construir estruturas de decisão. Os operadores são `&&`, `||`. Veja alguns exemplos:

```
if ($x > 0) { print $x; }       # pode ser escrito como:
$x > 0 && print $x;

unless ($y >= 1) { print $y; }  # pode ser escrito como:
$y >= 1 || print $y;
```

Nas expressões `&&`, o primeiro termo é executado. Se retornar *falso*, o segundo é ignorado e a expressão completa retorna *falso*. Se a expressão retornar *verdadeiro*, o valor final da expressão dependerá da execução do segundo termo. Nas expressões `||` se o primeiro termo for *verdadeiro*, o segundo é ignorado e o valor retornado será *verdadeiro*. Se for *falso*, é preciso executar o segundo que dirá o valor da expressão.

O operador ternário `?:` pode ser usado para substituir o `if-else`:

```
if ($x > 0) { $x++; } else { $x--; } # pode ser escrito como:
($x > 0) ? $x++ : $x--;
```

1.10. Vetores associativos

Vetores associativos são coleções de escalares assim como os vetores comuns, mas eles não têm uma ordem definida. Os seus valores são organizados em pares. Um deles é chamado de chave e é utilizado para recuperar o outro valor.

Uma variável de vetor associativo tem um identificador que inicia com um sinal de percentagem `%`. Cada elemento de um vetor associativo é uma variável escalar distinta, acessada por uma outra variável escalar, chamada de chave (*key*). Os elementos são localizados ou criados fornecendo-se a chave entre chaves `{` e `}`:

```
$elem{"Rússia"} = "Moscou";
$elem{"China"} = "Beijing";
$z = $elem{"Índia"}; #retorna o valor armazenado na chave "Índia"
```

Se o objeto não existir, o valor `undef` é retornado.

Perl não possui uma representação uniforme para um vetor associativo (não é possível controlar a ordem dos elementos). Na criação e na impressão, o vetor associativo é representado como uma lista ou vetor, com um número par de elementos. A ordem desse vetor, porém, não é definida (pode não ser sempre a mesma). Embora seja possível copiar um vetor associativo para um vetor indexado, não há como saber qual elemento será colocado em que posição.

```
%pares = ("Rússia", "Moscou", "Índia", "Delhi", "Itália", "Roma");
@vetor = %pares;
```

1.11. Operadores de associações

É possível realizar operações em vetores associativos com os operadores `keys`, `values` e `each`. O operador `keys` retorna uma lista (vetor indexado) das chaves de um vetor associativo. `values` faz o mesmo com os valores. Se não houver elementos no vetor, uma lista vazia `()` será retornada.

```
%cores = ("vermelho","ff0000","azul","0000ff","verde","00ff00");
@chaves = keys %cores;          # ("vermelho", "azul", "verde")
@valores = values %cores;       # ("ff0000", "0000ff", "00ff00")
```

O operador `each` serve para iterar sobre cada um dos elementos de um vetor associativo. É bastante útil quando utilizado junto com um `for` ou um `while`:

```
while (($cor, $codigo) = each %cores) {
     print "<td bgcolor=\#$codigo>$cor</td>";
}
```

Para remover elementos de um vetor associativo, é preciso usar o operador `delete`:

```
delete $cores{'azul'};
```

1.12. Operador eval

O operador `eval` interpreta uma *string* como código *Perl*. É útil para construir instruções e outras operações para posterior execução:

```
$x = "13.2";
$y = "19.6";
```

```
$op = "+";
eval("$x $op $y"); # resolve a expressão 13.2 + 19.6
```

2. Entrada e saída

Nesta seção apresentaremos os mecanismos básicos para entrada e saída de dados em um programa *Perl*, como leitura da entrada padrão, gravação na saída padrão, controle do sistema de arquivos e leitura e gravação em arquivos.

2.1. Entrada e saída padrão

Nos primeiros exemplos apresentados, mostramos exemplos de leitura de uma linha da entrada padrão usando o operador <STDIN>. Passando esse operador para uma variável escalar, obtém-se a próxima linha da entrada ou undef, se não houver mais linhas:

```
$linha = <STDIN>; #lê próxima linha da entrada padrão
```

Se você atribuir <STDIN> a um vetor, todas as linhas restantes serão lidas e armazenadas em um vetor indexado (lista). Cada linha ocupará o espaço de um elemento, inclusive a nova-linha "\n". Para arrancar as novas linhas de todos os elementos do vetor, pode-se usar chop.

```
@linhas = <STDIN>; # lê todas as linhas da entrada padrão
chop @linhas; #arranca últimos letras de todos os elementos de @linhas
```

Uma forma comum de ler todas as linhas da entrada padrão é fazer isto dentro de um bloco de repetição while:

```
while ($_ = <STDIN>) {
    chop $_;
    #fazer alguma coisa com $_;
}
```

Essa operação é tão comum que não é preciso usar a atribuição $_ = <STDIN>. Sempre que um *loop* contém apenas um operador de entrada de dados <...>, o seu conteúdo é armazenado na variável especial $_. O bloco acima, portanto, também pode ser escrito da seguinte maneira:

```
while (<STDIN>) {
    chop $_;
    #fazer alguma coisa com $_;
}
```

Para ler o conteúdo de arquivos passados na linha de comando, preenchidas no momento da execução do programa, utiliza-se o operador <>. Por exemplo, suponha o seguinte programa type.pl que é chamado da forma:

```
perl type.pl inuteis1.txt inuteis2.txt
```

Se você fizer:

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

todo o conteúdo dos dois arquivos será impresso na tela.

Os dados passados na entrada padrão também podem ser obtidos através do vetor `@ARGV`. Esse vetor contém todos os nomes que aparecem após o nome do programa que podem ser obtidos através de `$ARGV[0]`, `$ARGV[1]`, etc. A variável `$ARGV` (sem índice) contém o nome do programa.

Para ler um determinado número de caracteres da entrada padrão (ou de qualquer outro dispositivo) pode-se usar o operador `read`. O nome `STDIN` na verdade é apenas uma constante que serve de meio de acesso à entrada padrão. `read` recebe um vetor que deve conter a origem, o destino e o número de caracteres que devem ser lidos, nessa ordem. Para ler 15 caracteres da entrada padrão para dentro da variável `$cgc`, pode-se fazer:

```
read(STDIN, $cgc, 25);
```

Para imprimir na saída padrão pode-se usar os operadores `print` ou `printf`. O operador `print` pode receber um escalar ou um vetor. Os escalares passados no vetor são concatenados antes de impressos. Os resultados das três instruções abaixo é o mesmo:

```
print "Hello, world!"; # escalar
print ("Hello, world!"); # vetor de um elemento
print ("Hello", "world", "!"); #vetor de três elementos
```

Para imprimir texto formatado, *Perl* dispõe do operador `printf` que funciona da mesma maneira que o `printf` do *C*. A sintaxe inclui valores especiais como "`s`" para *strings*, "`d`" para decimais e "`f`" para ponto-flutuante. Suponha que você tenha as seguintes variáveis:

```
$nome = "Aristóteles";
$idade = 2383;
$pi = 3.14159265359;
```

Usando o operador `printf` da forma:

```
printf("%25s %5d anos. Número: %5.4f\n", $nome, $idade, $pi);
```

obtém-se a seguinte linha formatada (veja tamanhos dos campos em baixo):

```
                Aristóteles  2383 anos. Número:     3.1416
-------------------------------------------------------------
12345678901234567890123 12345                12345.1234
       25 caracteres      5 dígitos      5 dígitos . 5 decimais
```

O sinal de `%`, se usado dentro do *string* de `printf`, deve vir precedido de `\`, caso contrário, será utilizado para se referir à quantidade de caracteres de uma *string* (sufixo `s`) ou número de dígitos de um decimal (sufixo `d`). Os números de ponto-flutuante podem ser arredondados pelo número de casas especificada após o ponto no campo `%` com sufixo `f`.

A função `printf` permite *imprimir* dados formatados. Em vez de impressos, os dados podem ser atribuídos a uma variável usando `sprintf`. Veja um exemplo:

```
$tf = sprintf("%25s %5d anos. Número: %5.4f\n", $nome, $idade, $pi);
# coloca os dados formatados dentro da variável $tf
print $tf; # imprime os dados formatados (como printf)
```

2.2. Arquivos

Assim como STDIN fornece um meio de acesso à entrada padrão, pode-se criar em *Perl* outras constantes para ter acesso a arquivos. São chamados de *file handles* (alças de arquivo). Além de STDIN a linguagem fornece ainda STDOUT – a saída padrão, e STDERR – a saída padrão de erro. A operação print imprime por *default* na saída padrão. Para imprimir em outro dispositivo, basta informá-lo após o operador:

```
print STDERR "Erro: tente outra vez! \n";
print "Saída normal\n"; # mesmo que print STDOUT "Saída normal\n";
```

O dispositivo de padrão de saída também pode ser alterado usando o operador select:

```
select STDERR; # seleciona STDERR como nova saída padrão
print "Erro: tente outra vez!\n"; # mesmo que print STDERR "...";
select STDOUT; # default agora volta a ser STDOUT
```

Constantes de acesso a arquivos podem ser definidas pelo programador e utilizam-se de um identificador que não possui prefixo (como $, %, @ ou &). Para evitar conflitos com palavras reservadas do *Perl*, recomenda-se usar somente maiúsculas para definir tais constantes.

Para criar novas constantes de acesso a arquivos utiliza-se o operador open. A sintaxe é:

```
open(NOME_DA_CONSTANTE, "nome_do_arquivo_ou_dispositivo");
```

O nome do arquivo pode ser o nome de um dispositivo como um programa que oferece o serviço de e-mail. Depois de criado a constante, ela pode ser usada para se ler o arquivo. Não interessa o conteúdo. Programas em *Perl* podem ler tanto arquivos de texto (7-bit) como arquivos binários (8-bits). Para abrir o arquivo nomes.txt e imprimir todo o seu conteúdo, pode-se fazer:

```
open(ARQNOMES, "nomes.txt");
while (<ARQNOMES>) {     # lê uma linha do arquivo e coloca em $_
    print $_;
}
```

O processo de abertura do arquivo pode falhar. Se isto ocorrer, open retorna undef. Uma forma de evitar um erro caso o arquivo não seja aberto é colocar o open dentro de um if ou unless e só tentar usar a constante criada se open tiver sucesso.

```
if (open(ARQNOMES, "nomes.txt")) {
     ...
} else {
    print "Erro: não foi possível abrir nomes.txt";
}
```

Geralmente, quando isto ocorre, é preciso terminar o programa. *Perl* facilita o processo com o operador die, que, como print, imprime um *string* só que na saída padrão de erro (STDERR) e depois sai do programa sinalizando com um código de erro.

```
unless (open(ARQNOMES, "nomes.txt")) {
   die "Erro: não foi possível abrir nomes.txt";
}
```

Isto também pode ser feito com o operador ||, da forma mais compacta ainda *open or die* (ou abre ou morre):

```
open(ARQNOMES, "nomes.txt") || die "Erro: não pude abrir nomes.txt";
```

Para abrir um arquivo para gravação é preciso indicar a operação adicionando um prefixo ao nome do arquivo. A gravação pode ser para sobrepor (substituir) os dados do arquivo (se ele existir) ou para acrescentar, preservando o que já existe. Em ambos os casos, se o arquivo não existir, ele será criado. O símbolo ">" deve ser usado para indicar gravação com sobreposição e ">>" para indicar gravação sem sobreposição. Para imprimir no arquivo, usa-se `print` com dois argumentos, sendo o primeiro o nome da constante de acesso ao arquivo:

```
open(ARQNOMES, ">>nomes.txt") || die "Erro: não pude abrir
                                      nomes.txt para gravação";
print ARQNOMES ("Bento Carneiro;", "Rua dos Vampiros Brasileiros, 27;");
print ARQNOMES "\n";
print "Nome cadastrado no banco de dados"; # imprime na saída padrão
```

O trecho de código abaixo lê uma linha de cada vez do arquivo `fonte.txt` e copia a linha ao arquivo `destino.txt`.

```
open(FONTE, ">fonte.txt") || die "Erro: não pude abrir fonte.txt";
open(DESTINO, ">destino.txt") || die "Erro: não pude abrir
                                      destino.txt para gravação";
while (<FONTE>) {
    print DESTINO $_;
}
```

Para fechar um arquivo, usa-se o operador `close`. Em geral, o sistema se encarrega de fechá-lo quando sai do *loop*, mas, caso seja necessário realizar o fechamento antes, use:

```
close FONTE;
close DESTINO;
```

É possível realizar diversos testes sobre os arquivos para obter informações sobre eles usando operadores *-op*, onde *op* pode ser uma entre mais de 20 letras disponíveis para testas se o arquivo é executável, se ele pode ser lido, se é uma conexão de rede, se é texto, se é diretório, sua idade, etc. A sintaxe típica é:

```
if (-op $arquivo) {
    # operações se -op for verdadeiro ....
}
```

Por exemplo `-e` serve para testar se um arquivo existe e `-d` retorna um valor verdadeiro se o arquivo for um diretório, então:

```
$arquivo = "papainoel";
if (-e $arquivo && !(-d $arquivo)) {
    # operações se -op for verdadeiro ....
}
```

só será executado se o `papainoel` existir e *não* for um diretório.

A tabela abaixo relaciona os principais operadores de teste de arquivos de *Perl* e seus significados. A maioria retorna uma condição verdadeira ou falsa. Outros retornam um valor numérico.

TESTE	SIGNIFICADO
`-r`	Arquivo ou diretório é acessível (pode ser lido)
`-w`	Arquivo ou diretório pode ser alterado
`-x`	Arquivo ou diretório é executável

TESTE	SIGNIFICADO
-e	Arquivo ou diretório existe
-z	Arquivo existe e tem tamanho zero
-s	Arquivo ou diretório existe e tem tamanho diferente de zero
-f	Operando é um arquivo comum
-d	Operando é um diretório
-l	Operando é um vínculo simbólico
-S	Operando é um soquete de rede
-c	Operando é um dispositivo especial
-T	Arquivo é ASCII
-B	Arquivo é BINARY
-M	Idade do arquivo em dias
-A	Último acesso do arquivo em dias
-C	Última modificação do arquivo em dias

Para mover (mudar o nome) e remover um arquivo ou diretório, *Perl* oferece dois operadores: `unlink` – remove um arquivo, e `rename` – altera o nome de um arquivo.

```
print "Digite o nome do arquivo a ser removido: ";
chop($arquivo = <STDIN>);
unlink ($arquivo);

unlink ("senhas.txt","jegue.gif");
unlink (<*.bak>); #isto é um 'glob'. Veja próxima seção

rename ("despesas.txt", "receita.txt") || die "Isto é ilegal!";
```

2.3. Diretórios

Varios operadores em *Perl* servem para acesso a diretórios. O mais simples é o `chdir`, que permite mudar de um diretório para outro, e navegar no sistema de arquivos:

```
chdir ("/docs/lixo") || die "Não é possível mudar para o lixo!"
```

O comando é dependente de plataforma. Para rodá-lo em *Windows*, é preciso usar a barra invertida a não ser que o programa seja criado para operar como CGI em um servidor *Apache* que, mesmo em *Windows*, interpreta a barra como separador de diretórios. Para criar e remover diretórios, *Perl* oferece respectivamente os operadores `mkdir` e `rmdir`. Na criação de diretórios usando `mkdir`, deve-se informar o formato de permissões *chmod* do *Unix*. Só é possível a remoção de diretórios vazios:

```
mkdir ("zona", 0777); #cria um diretório onde todos têm acesso total
mkdir ("casa", 0755); #cria um diretório onde o dono tem acesso total
                        mas o grupo e outros só podem ler e executar
rmdir("encosto") || die "Não é possível se livrar do encosto!";
```

O recurso do *Unix* de permitir a filtragem de listas de arquivos através do *Shell* é chamada de *globbing*. Isto pode ser usado em *Perl* colocando o padrão de busca entre sinais < e >:

```
@lixo = </docs/*.bak>;
```

Da forma acima, o *glob* retorna uma lista (vetor) contendo todos os nomes que coincidem com o padrão de busca. No contexto escalar (se o *glob* for atribuído a uma variável escalar), o próximo nome de arquivo será recebido.

Para ler o conteúdo de um diretório é preciso abri-lo e para isto, assim como fizemos com os arquivos, precisamos definir uma constante (*file handle*) que permita o acesso ao diretório. O operador `open` não deve ser usado neste caso mas *Perl* oferece um operador chamado `opendir` que tem exatamente a função de abrir diretórios e retornar uma constante que permita manipulá-lo. Uma vez aberto, o conteúdo do diretório pode ser listado com `readdir`.

```
opendir (CAIXA, "/pandora") || die "Você não pode abrir esta caixa!;
while ($males = readdir(CAIXA)) {
   print "$males\n";
}
```

Para sair do diretório, usa-se `closedir` (que geralmente não é necessário, pois o sistema fecha os arquivos e diretórios quando sai do *loop*.

```
closedir(CAIXA);
```

3. Expressões regulares

Expressões regulares representam padrões que devem ser comparados a uma cadeia de caracteres. O resultado da comparação pode ser utilizado simplesmente para tomar decisões (em caso de sucesso ou falha), para obter informações sobre o texto (como, quantas vezes e onde a comparação ocorreu) ou para realizar substituições baseadas no resultado da comparação.

Expressões regulares são comuns em vários programas e ferramentas de sistemas operacionais *Unix* como os *shells* e aplicações como *grep*, *awk*, *emacs* e *vi*. *Perl* possui o conjunto mais completo de expressões regulares entre essas ferramentas *Unix*. Como elas fazem parte da linguagem, também estão presentes nas implementações de *Perl* em *Windows*. Inspiradas em *Perl*, várias outras linguagens modernas, como *JavaScript*, adotam a sintaxe de expressões regulares usadas na linguagem, que hoje também está presente em ferramentas de busca e substituição de processadores de texto, programas de editoração eletrônica e ferramentas de desenvolvimento Web.

3.1. Sintaxe básica

A expressão regular é geralmente representada entre barras "/". Para usá-la, ela deve ser comparada com uma *string* que deve estar armazenada em uma variável escalar (pode ser `$_`) por exemplo:

```
$texto = "Os camelos foram beber água na cachoeira.";
if ($texto =~ /camelo/) {   # expressão regular em negrito!
    print "Existem camelos no texto!";
}
```

O operador =~ define o objeto no qual a expressão regular é calculada. Pode ser qualquer variável ou um dispositivo como `<STDIN>`. Se o objeto e operador =~ é omitido, a operação ocorre sobre a variável global `$_`.

```
$_ = "Os camelos foram beber água na cachoeira.";
if (/camelo/) {
    print "Existem camelos em todo lugar!";
}
```

Geralmente, expressões regulares são mais complexas do que as mostradas acima. Podem conter vários operadores, códigos e caracteres especiais que permitem uma busca mais refinada. Por exemplo:

```
if ($texto =~ /ca.el*o/) {
    print "Existem camelos, cabelos, camellos, cabellllos, ...!";
}
```

A busca com substituição requer expressões mais elaboradas que utilizam os operadores de *transliteração* e de *substituição*. Esses operadores podem ser aplicados para localizar padrões no texto e substituí-los por outros padrões, variáveis, *strings* ou pelo resultado de certas expressões. Contêm várias opções que permitem buscas e

substituições globais, com execução de expressões, etc. Serão apresentados no final desta seção. O exemplo a seguir substitui todas as ocorrências de "cavalo" por "jumento" em `$sitio`:

```
$sitio =~ s/cavalo/jumento/g
```

O sufixo `g` indica que a busca é global.

3.2. Mudança de delimitadores

O delimitador usado por *default* para expressões regulares é a barra "/". Se o texto contido entre barras possui barras, é preciso preceder cada uma delas com uma contra-barra (para anular o efeito da barra). Quando há muitas ocorrências de "/", o código pode ficar ilegível, pois cada "/" se transforma em "\/". É possível, porém, trocar o delimitador por outro como parênteses, chaves, colchetes, cifrões, etc. desde que seja *um* (ou *um par*) de caracteres *não-alfanuméricos*. Para fazer a mudança, o novo delimitador deve vir precedido pelo caractere `m`. O símbolo que seguir o `m` será o novo delimitador.

```
$caminho =~ /\/home\/usuarios\/helder\/public_html\/perl\//; #default: /
$caminho =~ m@/home/usuarios/helder/public_html/perl/@; # delim: @
$caminho =~ m(/home/usuarios/helder/public_html/perl/); # delim: (
```

Caso sejam usados os caracteres "(", "{" ou "[", o caractere de fechamento será o ")", "}" ou "]" correspondente.

Para substituições, transliterações e outras operações semelhantes (seção seguinte), que usam um operador (`s, tr`) com texto entre *três* delimitadores, o `m` é desnecessário. O delimitador desejado deve ser definido logo após o `s` ou `tr`:

```
$x =~ s/\/home\/usuarios\/helder\//\/~helder\//;   # default
$x =~ s(/home/usuarios/helder/)(/~helder/);      # delim: (
$x =~ s#/home/usuarios/helder/#/~helder/#;       # delim: #
$y =~ tr/abcd/ABCD/;               # delimitador default: /
$y =~ tr#abcd#ABCD#;               # delimitador: #
```

A troca de delimitadores só deve ser realizada nas poucas situações onde, de fato, melhora a legibilidade do código. Evite usar como delimitadores caractere que aparecem dentro da expressão.

3.3. Variáveis

O conteúdo de variáveis pode ser passado para expressões regulares da mesma forma como são interpretados dentro de *strings*. É possível, portanto, construir uma expressão regular a partir de *strings* previamente calculados:

```
$texto = "Os camelos passaram a noite numa festa com as llamas.";
print "Digite uma palavra para procurar no texto: ";
$palavra = <STDIN>;
chop $palavra;
if ($texto =~ /\b$palavra\b/) {
    print "A palavra \"$palavra\" foi encontrada no texto!";
} else {
    print "A palavra \"$palavra\" não foi encontrada no texto!";
}
```

O símbolo `\b` é utilizado para marcar os limites de uma palavra (para evitar que a palavra "aspa", por exemplo, combine com "caspa" ou com "aspargos").

3.4. Padrões simples

A comparação mais simples é aquela que procura combinar uma seqüência exata. Por exemplo, encontrar o caractere "a" em uma *string* ou a palavra "camelo". Um exemplo desse tipo de combinação foi mostrado na seção anterior.

O caractere "`.`" (ponto) é utilizado para representar qualquer caractere, com exceção do caractere de nova-linha ("`\n`"). No exemplo mostrado anteriormente, `/ca.elo/` combina com "camelo", "cabelo", "caeelo", "ca&elo", etc.

Pode-se restringir os caracteres permitidos usando classes de caracteres, representadas por uma seqüência de caracteres permitidos entre colchetes `[` e `]`. Somente um dos caracteres da lista pode estar presente no lugar marcado. Por exemplo, `/ca[bmp]elo/` combina com "camelo", "cabelo" e "capelo", mas não com "caBelo" ou "ca_elo". Para representar uma seqüência de caracteres, pode-se usar o caractere "`-`", por exemplo:

```
/[abcdeABCDE]/   # é a mesma coisa que /[a-eA-E]/
/[0-9]/          # é a mesma coisa que /[0123456789]/
```

Pode-se também inverter a seleção, ou seja, permitir qualquer caractere que não esteja na lista. Para isto, basta colocar um circunflexo "^" antes da seqüência:

```
/[^abcdeABCDE]/ # qualquer caractere menos "a", "b", "c", "d" e "e".
/[^0-9]/        # tudo menos números
```

Para incluir os caracteres "`-`", "^", "`\`" e "`]`" na lista, é preciso precedê-los por uma contra-barra.

Várias classes são predefinidas em *Perl* e suas seqüências podem ser substituídas por caracteres de escape, mostrados na tabela abaixo:

SEQÜÊNCIA	CLASSE EQUIVALENTE
`\d`	`[0-9]`
`\w`	`[a-zA-Z0-9_]`
`\s`	`[ \r\t\n\f]`
`\D`	`[^0-9]`
`\W`	`[^a-zA-Z0-9_]`
`\S`	`[^ \r\t\n\f]`

3.5. Seqüências, multiplicadores e alternação

Os símbolos usados na seção anterior permitem combinar no máximo um caractere. Para fazer combinações mais complexas, é preciso usar outros padrões chamados de seqüências, multiplicadores e alternação. A seqüência já foi mostrada anteriormente. Consiste de qualquer seqüência de caracteres entre barras, como `/camelo/`. Para representar zero ou mais ocorrências de um determinado caractere, utiliza-se o mais simples dos multiplicadores, o asterisco "`*`". Além dele existem o sinal de mais "+" que representa um ou mais (o caractere anterior), e o ponto de interrogação "?", que significa zero ou uma ocorrências do caractere anterior. A tabela abaixo fornece um resumo desses símbolos:

MULTIPLICADOR	MULT. GERAL	RESULTADO
`c*`	`c{0,}`	zero ou mais "c"

MULTIPLICADOR	MULT. GERAL	RESULTADO
`c+`	`c{1,}`	um ou mais "c"
`c?`	`c{0,1}`	zero ou um "c"

Por exemplo, a expressão regular `/ca+me*los?/` combina com "camelo", "camelos", "cameeelos" e "caaamlo", mas não com "cmelo" ou "cameloss".

A substituição usando multiplicadores é sempre realizada com o máximo de elementos possível que combine com o padrão. Por exemplo, a expressão regular de substituição

```
s/a*h/u,
```

quando aplicada à *string*

```
$_ = "Baaaaaaaaah, tchê!";
```

substituirá todos os 10 a's consecutivos com uma única letra "u". Se a intenção é substituir grupos menores, é preciso usar o multiplicador geral, definido entre chaves `{` e `}`. Com ele, pode-se definir uma faixa de valores indicando a quantidade de elementos a serem substituídos. *Exemplos*:

```
s/a{5}h/u    # exatamente 5 "a": Buuh, tchê!
s/a{2,4}h/u  # de 2 a 4 (de preferência) "a": Buuuh, tchê!
s/a{0,3}h/u  # de 0 a 3 "a"
s/a{6,}h/u   # 6 ou mais "a"
```

Para combinar com uma lista de alternativas possíveis de caracteres, pode-se usar as classes de caracteres (entre colchetes). Para combinar entre alternativas de seqüências, porém, as classes não servem. Para isto, usa-se um grupo chamado de alternação, onde as alternativas são separadas pelo caractere "`|`".

```
/dia|tarde|noite|madrugada/ # combina com uma das alternativas
/(segunda|terça|quarta|quinta|sexta)-feira|sábado|domingo/
# parênteses definem precedência. Opções serão segunda-feira,
# terça-feira, ..., domingo.
```

3.6. Âncoras

Há quatro caracteres especiais que ancoram um padrão. Pode-se identificar o início da *string*, o final e a exigência ou não de um limite (início ou final) de uma palavra.

A tabela abaixo relaciona os quatro tipos de padrão.

ÂNCORA	FUNÇÃO
`^`	Início da cadeia de caracteres
`$`	Fim da cadeia de caracteres
`\b`	Fim ou início de palavra
`\B`	Não há fim ou início de palavra

O marcador "`\b`" marca o início ou o final de uma palavra. Já o marcador marca a inexistência desse limite. Veja alguns exemplos:

```
/pão\b/ # combina com pão mas não com pãozinho
/\bpão/ # combina com pão e pãozinho mas não com sapão
```

```
/\bpão\b/ # combina com pão mas não com pãozinho nem sapão
/\bpão\B/ # combina com pãozinho mas não com pão francês
/\Bpão\B/ # combina com sapãozinho mas não com um pão francês
```

Para representar o início de uma *string*, usa-se o circunflexo "^". Este caractere só tem significado especial quando aparece no início da expressão regular. Em qualquer outro lugar, representa o caractere circunflexo. Se for necessário localizar o caractere circunflexo no início do padrão, deve-se precedê-lo por uma contra-barra.

```
/^morcego^/   # combina com morcego^ no início da string
/\^morcego^/  # combina com ^morcego^ em qualquer lugar
```

O cifrão representa o final da cadeia de caracteres. Só faz sentido usá-lo no final de um a expressão (em qualquer outro lugar será interpretado como o prefixo de uma variável escalar).

```
/R$/   # combina com R no fim da string
/R\$/  # combina com R$ em qualquer lugar da string
```

Quando houver vários tipos diferentes de operadores de grupo (multiplicadores, âncoras, alternação, seqüências) sempre pode-se usar parênteses para resolver o problema de operações que devem ser realizadas antes de outras. Há uma ordem de precedência para esses operadores que está mostrada na tabela abaixo:

	OPERADORES	TIPO
1	( )	parênteses
2	+ * ? {m,n}	multiplicadores
3	abcde	seqüências
	^ $ \b \B	âncoras
4	|	alternação

Veja abaixo alguns exemplos de expressões regulares e o efeito dos parênteses e das regras de precedência:

```
/a|b*/        # um único a ou seqüência de bbbbb...
/(a|b)*/      # seqüência de aaa... ou de bbb...
/abc*/        # ab, abc, abcc, abccc, abcccc ...
/(abc)*/      # "", abc, abcabc, abcabcabc, ...
/^x|y/        # x no início da string ou y em qq. lugar
/^(x|y)/      # x ou y no início da string
/a|bc|d/      # a ou bc ou d
/(a|b)(c|d)/ # ac, ad, bc ou bd
```

Importante: Não confunda `/^abc/` com `/[^abc]/`. A primeira expressão combina com a *string* "abc" no início da linha. A segunda combina com *um caractere* apenas que *não* seja "a", "b" ou "c".

3.7. Parênteses como memória

Além de controlar a precedência das operações de grupo, parênteses também são usados como memória para armazenar o resultado de expressões para uso posterior. Para recuperar um padrão previamente armazenado, é preciso utilizar uma contra-barra seguida por um inteiro (`\1`, `\2`, `\3`, etc.) representando a ordem em que a expressão (ou substring) ocorre. Cada par de parênteses utilizado, mesmo se utilizado apenas para determinar precedência, é lembrado e recuperado por um número inteiro. Veja alguns exemplos

```
/<(b)>.*<\/\1>/ # combina com qualquer coisa entre <b> e </b>
/25(.)12\1999 13(:)15\259/ # 25-12-1999 13:15:59 ou 25/12/1999 ...
/C(.*)s e D\1s/ # Cobras e Sobras, Cucos e Sucos, Cantos e Santos
```

As principais aplicações dos parênteses como memória ocorrem em substituições:

```
$pagina = "<html><head> ... </body></html>";
$pagina =~ s/<([^>]*.)>/&lt;\1&gt;/g
# troca todos os <...> ('...' ñ contém ">") por &lt;...&gt;
```

Depois de uma combinação bem sucedida (como o exemplo acima), as variáveis somente-leitura `$1`, `$2`, `$3`, etc. são definidas com os mesmos valores de `\1`, `\2`, `\3`, etc. Elas podem então ser usadas em outras partes do programa.

```
$bichos = "As cobras e os ratos estão com sede.";
$bichos =~ /.s c(.*)s e .s r(.*)s /; # observe o espaço após o "s"
print "Os g$2s devoram as s$1s do restaurante\n";
```

As variáveis também podem ser lidas em um contexto de vetor:

```
$_ = "As cobras e os ratos estão com sede.";
($primeiro, $segundo) = /.s c(.*)s e .s r(.*)s /;
# ou @valores = /.s c(.*)s e .s r(.*)s /;
```

No exemplo acima, `$primeiro` contém o valor armazenado em `$1` e `$segundo` contém o valor em `$2`. Há ainda outras três variáveis predefinidas que obtém informações adicionais sobre a expressão regular. Todas estão mostradas na tabela abaixo.

VARIÁVEL	FUNÇÃO
`$1, $2, $3, ...`	Guardam os valores armazenados nos parênteses
`$&`	Guarda o resultado da expressão regular
`$``	Guarda o texto antes do resultado (crase)
`$'`	Guarda o texto após o resultado (apóstrofe)

As variáveis são definidas após cada expressão calculada. Se for necessário utilizá-las mais adiante no programa, deverão ser copiadas para outras variáveis. Veja um exemplo:

```
$texto = "Os camelos passaram a noite numa festa com as llamas";
$texto =~ /ca(.el)*o/;
print "Primeiro texto armazenado: $1\n"; # imprime "mel"
print "Texto localizado: $&\n"; # imprime "camelos"
print "Texto anterior: $'\n";  # imprime "Os "
print "Texto posterior: $`\n";  # imprime " passaram ... llamas"
```

3.8. Outros operadores

O sufixo `i` (de *ignore case*) pode ser acrescentado após a última barra de uma expressão regular para torná-la insensível ao formato maiúsculo ou minúsculo dos caracteres (*case-insensitive*). Veja um exemplo:

```
print "Digite o que você quer fazer: ";
$nome = chop <STDIN>;
if ($nome =~ /^listar/i) {
```

```
        # Usuário pode digitar "Listar" ou "LISTAR RESULTADOS" ou ...
    }
```

3.9. Substituições

O operador de substituição, representado pelo prefixo `s`, colocado antes de uma expressão regular de duas partes, tem a seguinte sintaxe básica:

```
s/texto_antigo/texto_novo/
```

A expressão acima altera apenas a primeira ocorrência de `texto_antigo`. Para localizar e substituir todas as ocorrências, utiliza-se o sufixo `g`:

```
s/texto_antigo/texto_novo/g
```

O sufixo e é usado para tratar a segunda parte de um substituição como uma expressão. Na substituição, a segunda parte é executada:

```
s/(camelos)/chop $1/e   # substitui camelo por camelos
```

A tabela abaixo contém vários sufixos freqüentemente usados nos operadores de substituição. Os sufixos podem ser colocados lado a lado em qualquer ordem. O resultado de qualquer uma das duas expressões abaixo é o mesmo:

```
s/texto_antigo/chop($texto). !/ge;
s/texto_antigo/chop($texto). !/eg;
```

SUFIXO	FUNÇÃO
`i`	Ignora formato caixa-alta ou caixa-baixa em qualquer expressão regular.
`e`	Trata a segunda parte da substituição como expressão *Perl* (e executa).
`g`	Realiza a substituição em todo o *string* (busca e substituição global)

3.10. Transliteração

O operador de transliteração é útil para realizar substituições simultâneas entre caracteres. Para realizar uma transliteração, usa-se o comando `tr`, seguido de dois argumentos entre barras (ou outro delimitador qualquer), assim como o comando `s`. Os operandos consistem de uma seqüência antiga e uma nova. O primeiro caractere da nova seqüência substituirá todas as ocorrências do primeiro caractere da antiga seqüência encontrados no texto-objeto da substituição (ou a variável `$_`). O mesmo ocorrerá com os segundos caracteres, os terceiros, etc.

```
$texto = "O rei roeu os rabos dos ratos de Roma.";
$texto =~ tr/aeiou/AEIOU/;
# $texto agora contém "O rEI rOEU Os rAbOs dOs rAtOs dE ROmA."
```

É possível especificar uma seqüência ordenada usando o hífen. Por exemplo, tanto faz usar `0123456789` ou `0-9`. Se a primeira seqüência tiver mais caracteres que a segunda, o último caractere da segunda seqüência será repetido para cada caractere da primeira seqüência que não tiver um correspondente na segunda. Veja alguns exemplos:

```
$texto = "O rato roeu a roupa do rei de Roma.";
$texto =~ tr/aeiou/AEIOU/; # 26 cars. para 5
# $texto agora contém "O UAUO UOEU A UOUUA UO UEI UE ROUA."
```

Pode ser usado o valor de retorno de uma transliteração, que contém o número de caracteres afetados. Para obter esse valor, basta atribuir o resultado da expressão à uma variável:

```
$_ = "O rei roeu os rabos dos ratos de Roma.";
$num = tr/aeiou/AEIOU/; # num contém 14 e
# $_ contém "O rEI rOEU Os rAbOs dOs rAtOs dE ROmA."
```

É possível usar o operador `tr` para obter o número de caracteres que combinam com a primeira expressão, sem alterar a *string*, mantendo o segundo argumento da operação vazio:

```
$_ = "O rei roeu os rabos dos ratos de Roma.";
$num = tr/aeiou//; # num contém 14 e $_ não foi alterado.
```

Na tabela abaixo estão as principais opções que podem modificar o operador `tr` e sua função. Assim como as opções de `s`, elas são listadas depois da segunda seqüência, em qualquer ordem:

```
$texto =~ tr/aeiou/AEIOU/cds;
```

SUFIXO	FUNÇÃO
d	Remove caracteres que não aparecem na segunda parte da transliteração.
c	Resultado é conjunto de todos os caracteres que não aparecem na primeira parte da transliteração.
s	Converte múltiplas cópias seguidas de uma letra na segunda parte de uma transliteração como uma só.

A melhor forma de ilustrar o funcionamento de `tr` é através de exemplos. Considere a seguinte *string*:

```
$_ = "O rei roeu os rabos dos ratos de Roma.";
```

A opção `d` elimina quaisquer caracteres listados na primeira seqüência que não têm correspondente na segunda. Abaixo, apenas `aeiou` combina. Sem a opção `d`, a letra `U` substituiria cada letra não encontrada.

```
tr/aeiou/bcdfghjklmnpqrstvwxyz/AEIOU/d; #apaga caracteres
# $_ agora contém "O EI OEU O AO O AO E ROA."
# (os caracteres a-z exceto vogais foram eliminados)
```

A opção `c` considera o complemento (a negação) da primeira seqüência em relação ao universo de caracteres (256) disponíveis.

```
tr/a-z/_/; # sem opções
# $_ agora contém "O ___ ____ __ _____ ___ _____ __ R___."
tr/a-z/_/c; #substitui complemento (caracteres não listados)
# $_ agora contém "__rei_roeu_os_rabos_dos_ratos_de__oma_"
# (todos os caracteres exceto a-z (O,R, ,.) foram convertidos em _
```

A opção `s` ignora repetições sucessivas de uma mesma letra.

```
tr/aeiou/x/;  # sem opções
# $_ agora contém "O rxx rxxx xs rxbxs dxs rxtxs dx Rxmx."
$_ = "O rei roeu os rabos dos ratos de Roma.";
tr/aeiou/x/s; # ignora caracteres repetidos
```

```
# $_ agora contém "O rx rx xs rxbxs dxs rxtxs dx Rxmx."
$_ = "O rei roeu os rabos dos ratos de Roma.";
tr/a-z/x/s; # ignora caracteres repetidos
# $_ agora contém "O x x x x x x x Rx."
```

É possível fazer substituições sofisticadas combinando mais de uma opção:

```
$_ = "um     dois   três  quatro        cinco.";
tr/a-zA-Z\./ /cs; # somente espaços, ignora repetições
# $_ agora contém "um dois tr s quatro cinco."; # ê ñ está entre a-z

$_ = "um     dois   três  quatro        cinco.";
tr/a-zA-Z\.ê/ /cd; # elimina todos os espaços
# $_ agora contém "umdoistrêsquatrocinco.";
```

3.11. Operadores split e join

O operador `split` recebe uma *string* e uma expressão regular. Ele faz uma busca global pela expressão e retorna um vetor contendo as partes da *string* que não combinam com a expressão. O resultado da expressão regular serve, então, como um separador para transformar *strings* em vetores:

```
$registro = "nome; endereço; telefone; email";
@campos = split(/;/, $registro);
($nom, $end, $tel, $ema) = @campos;
```

O valor *default* para o operador `split` é "um ou mais espaços consecutivos", ou seja `/\s+/`. O objeto *default* é a variável `$_`. As duas operações a seguir são, portanto, equivalentes:

```
$_ = "um   dois         três    quatro                  cinco";
@nums = split(/\s+/, $_);  # eh a mesma coisa que...
@nums = split; # resultado: ("um","dois","três","quatro","cinco")
```

O operador `join` é o contrário do `split`. Enquanto aquele separa, este junta. Os argumentos são: um vetor que terá seus elementos concatenados e uma *string*, que será repetida entre os elementos do vetor.

```
$linTab = join("</td><td>", @nums);
# res.: um</td><td>dois</td><td>três</td><td>quatro</td><td>cinco
print "<table border=1><tr><td>".$linTab."</td></tr></table>";
```

4. Subrotinas e bibliotecas

4.1. Subrotinas

O operador `sub` é utilizado em *Perl* para definir um procedimento ou subrotina que pode ser chamado de outra parte do programa ou de outro programa (se incluído em uma biblioteca). Por exemplo:

```
sub soma {
   print 2+2;
}
```

define a subrotina soma. Para chamá-la, utiliza-se um identificador com o nome da subrotina precedido do caractere "`&`".

```
&soma; # imprime 4.
```

Uma subrotina pode ter qualquer número de instruções, simples ou compostas. O último valor resultante de uma expressão é sempre retornado. Por exemplo, na subrotina:

```
sub soma {
   2+2;
}
```

a chamada

```
$res = &soma;
```

armazena o valor 4 em `$res`.

Uma subrotina termina assim que chega ao fim. Para escapar antes, pode-se usar o operador `return` que também pode ser usado para devolver um valor, ao deixar a operação:

```
sub soma {
   return 2+2;
}
```

O valor de retorno não precisa ser um escalar. Pode também ser um vetor.

Algumas subrotinas podem requerer parâmetros passados na sua chamada. Uma rotina de soma mais útil seria chamada da forma:

```
$res = &soma(13,4);
```

O argumento que segue a chamada da função é um vetor. Ele será automaticamente armazenado no vetor global `@_`. Dentro da subrotina, os valores passados podem ser recuperados lendo os valores desse vetor:

```
sub soma {
   ($x, $y) = @_;
   return $x + $y;
}
```

Dentro da subrotina, `@_` é local. As outras variáveis definidas dentro da subrotina, porém, não são. Para declarar variáveis como locais é preciso usar o operador local. A forma típica de construir funções que recebem parâmetros em *Perl*, é, portanto:

```
sub soma {
   local($x, $y, @extras) = @_; #$x e $y são locais (como @_)
   # resto da função.
}
```

A variável `@_` também pode ser referenciada em partes escalares, utilizando os identificadores `$_[0]`, `$_[1]`, etc. O uso de elementos do vetor `@_` que não existem não provoca erros (todos contém `undef`).

O seguinte exemplo mostra uma subrotina que pode aceitar qualquer número de argumentos.

```
sub somatorio {
   local ($sum);
   $sum = 0;
   foreach $_ (@_) {
      $sum += $_;
   }
   return $sum;
}
```

Para chamar a função acima, pode-se usar qualquer vetor ou expressão que resulte em vetor, para passar os parâmetros após o nome da função:

```
&somatorio(4, 8, 10); # soma 4 + 8 + 10
&somatorio(1..20);    # soma 1+2+3+...+19+20
```

4.2. Bibliotecas

Trechos de código em *Perl* definidos em outros arquivos podem ser incluídos em um programa usando a instrução `require`. Esse recurso é extremamente útil para a definição de bibliotecas de subrotinas e variáveis globais que devem ser compartilhadas por mais de um programa.

Qualquer programa em *Perl* pode ser incluído em outro usando `require`. É preciso, porém, que a última linha do código incluído contenha um valor definido e positivo (o valor é retornado pelo `require`). Tipicamente, os arquivos a serem incluídos consistem de uma coleção de subrotinas (blocos `sub { ... }`). Depois das subrotinas, deve haver mais uma linha no fim do arquivo:

```
1;
```

A linha acima é o suficiente para que o arquivo possa ser usado como biblioteca e importado com `require`:

```
require "biblio.pl";
```

Na distribuição original do *Perl* (desde a versão 4) há várias bibliotecas que podem ser importadas usando `require`. Tipicamente, os arquivos têm a extensão `.pl` ou `.ph`. A variável global `$INC[0]` (do vetor `@INC`)

contém o diretório onde tais bibliotecas estão armazenadas (nas distribuições padrão do *Perl* para *Unix* ou no *ActivePerl* para *Windows*).

Várias outras bibliotecas úteis podem ser encontradas na Internet. Uma das mais populares é a biblioteca `cgi-lib.pl`[1], que contém várias subrotinas úteis ao tratamento de dados em programas CGI. Outras como `oraperl.ph` e `sybperl.ph` contém subrotinas e variáveis úteis para o acesso a bancos de dados *Oracle* e *Sybase*, respectivamente. Também podem ser encontradas na Internet[2].

[1] Veja na Internet em http://www.cgi-lib.org. Há também bibliotecas para C e outras linguagens. Para manipulação de "cookies" via CGI, veja a biblioteca Perl cookie.lib em http://www.worldwidemart.com/scripts/

[2] http://src.doc.ic.ac.uk/packages/perl/db/perl4/. Veja também os módulos (orientados a objetos) do Perl 5.

5. Transformação de dados

Uma das principais vantagens de *Perl* como linguagem preferencial para uso na Web (em programas CGI) é a quantidade de recursos que possui para realizar transformação e formatação de dados. Nesta seção serão apresentados os principais operadores para manipulação e formatação de *strings*.

5.1. Manipulação de texto

Três operadores existem para localizar e extrair trechos de um texto mais longo. As funções `index` e `rindex` retornam um número indicando a posição onde tem início uma determinada seqüência de caracteres dentro de um texto maior. A única diferença entre as duas funções é a ordem em que realizam a busca. `index` procura o texto do início até o fim da *string* e `rindex` faz a busca no sentido contrário. A sintaxe geral é:

```
$pos = index($str, $substring); # ou rindex($str, $substring);
```

O número retornado será um valor inteiro maior ou igual a zero e menor que o comprimento total da *string*. Se a *substring* não for encontrada, o valor retornado será `-1`. Se houver mais de uma ocorrência do texto procurado, apenas a posição do primeiro encontrado será retornada (que poderá ser diferente caso seja usado `index` ou `rindex`). É possível iniciar a busca a partir de uma posição específica, informada como terceiro argumento:

```
$pos = index($str, $subs, $ini); # ou rindex($str, $subs, $ini);
```

Veja alguns exemplos usando `index` e `rindex`:

```
$texto = "Abandonai toda esperança, ó vós que entrais!";
$num1 = index($texto, "an");          # $num1 contém 2
$num2 = index($texto, "an", $num+1); # $num2 contém 20
$num3 = rindex($texto, "an");         # $num3 contém 20
$num4 = rindex($texto, "an", $num);  # $num4 contém 2
```

A função `substr` extrai uma parte de um texto maior a partir de suas posições inicial e final. A sintaxe básica é:

```
$novoString = substr($string, $inicio, $fim);
```

As variáveis `$inicio` e `$fim` devem ser inteiros. Se `$início` for menor que zero, a contagem começa a partir do final da *string*. `$fim` deve ser maior que zero para que o resultado retorne uma *string* não-vazia. Veja alguns exemplos:

```
$frag = substr($texto, $num1, $num2); # ou substr($texto, 2, 20);
# $frag contém "andonai toda esper" (caracteres 2 a 19)
```

```
$fim = substr($texto, -8, 6); # contém "entrai"
# iniciando na 8a. letra a partir do fim, retornar as próximas 6
```

O operador `substr` também pode aparecer do lado esquerdo da operação de atribuição. Nessa situação, ele pode receber um novo texto que irá substituir a *substring* no texto original. O texto novo pode ser maior ou menor que o texto substituído:

```
$texto = "Abandonai toda esperança, ó vós que entrais!";
substr($texto, 0, 7) = "Deix"; # 4 letras no lugar de 7
# texto contém "Deixai toda esperança, ó vós que entrais!";
substr($texto, -8, 4) = "cheg";
substr($texto, 0, 4) = "Pass";
substr($texto, 12, 20) = "a grana";
# texto contém "Passai toda a grana, ó vós que chegais!";
```

5.2. Formatos

Um formato é um gabarito (*template*) de impressão. Permite criar um *layout* para posterior formatação de texto. Com formatos pode-se definir como o texto será posicionado na página e fixar previamente títulos de colunas, rótulos cabeçalhos e rodapés.

Antes que um formato possa ser usado ele precisa ser definido. Uma vez definido, pode receber os dados que serão posicionados em seus campos. Finalmente, ele pode ser chamado e impresso.

Um formato é definido através da palavra reservada `format`, seguida de um identificador que será o nome de um formato. Segue a definição do formato (que pode ocupar zero ou mais linhas). A definição termina com um ponto "`.`" isolado no início de uma linha. A definição contém um conjunto de *campos fixos* (linhas de texto), que contém valores fixos que devem ser impressos. Essas linhas podem também conter *campos variáveis* que são reservados através de operadores de formatação. Após uma linha que contém campos variáveis, deve haver outra linha contendo as variáveis que devem substituí-los separadas por vírgulas. Veja um exemplo de formato abaixo:

```
format ETIQUETA =
+-----------------------------------------------+
| Nome: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$nome
| Endereço: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$endereco
| Cidade: @<<<<<<<<<<<<<<< UF: @< CEP: @<<<<-@<<|
$cidade, $uf, $cep1, $cep2
+-----------------------------------------------+
.
```

O operador `@<<<` reserva espaço para um determinado número de caracteres alinhados pela esquerda. `@` representa um caractere, `@<` representa dois, e assim por diante. Se uma linha de campo fixa contiver 3 campos variáveis `@`, ela deve vir seguida de uma linha com 3 variáveis escalares separadas por vírgulas.

Uma vez definido o formato, ele pode ser chamado com o operador `write`. Antes ele deve ser preenchido com os dados que irá formatar. Isto é feito atribuindo valores às variáveis globais utilizadas pelo formato. Os dados podem ser lidos de fontes externas. Por exemplo, suponha um arquivo com vários registros separados por ponto-e-vírgula:

```
Marie Curie;Rua do Césio, 137;Vila Tchernobyl;PE;50213-320
Hans Staden;Rua Potiguares, 13;São Paulo;SP;01234-970
```

Se essas informações estão em um arquivo `dados.txt`, podem ser lidas pelo programa em *Perl* usando `open`. O resultado do formato deve ser redirecionado para um arquivo (ou qualquer dispositivo) que será aberto para gravação. O nome do descritor de arquivo deve ser o mesmo nome do formato:

```
open (ETIQUETA,">etiquetas.txt") || die "Não pude criar formato";
open (DADOS,"dados.txt") || die "Não pude abrir arquivo";
while (<DADOS>) {   # lê uma linha do arquivo para $_
   chop; # remove \n de $_
   ($nome, $endereco, $cidade, $uf, $cep) = split(/;/);
   ($cep1, $cep2) = split(/-/,$cep);
   write ETIQUETA; # envia variáveis lidas para formato
}
```

Depois do código acima, o arquivo `etiquetas.txt` deve conter:

```
+---------------------------------------------+
| Nome: Marie Curie                           |
| Endereço: Rua do Césio, 137                 |
| Cidade: Vila Tchernobyl  UF: PE CEP: 50213-320|
+---------------------------------------------+
+---------------------------------------------+
| Nome: Hans Staden                           |
| Endereço: Rua Potiguares, 13                |
| Cidade: São Paulo        UF: SP CEP: 01234-970|
+---------------------------------------------+
```

Esta seção apenas apresentou os conceitos básicos dos formatos *Perl*. Há vários outros operadores e maneiras diferentes de manipular, construir e utilizar formatos. A tabela abaixo oferece uma lista dos principais operadores para a definição dos campos de um formato.

OPERADOR	FUNÇÃO
`@<<<<`	Reserva espaço para 5 caracteres alinhados pela esquerda.
`@>>>>`	Reserva espaço para 5 caracteres alinhados pela direita.
`@||||`	Reserva espaço para 5 caracteres alinhados pelo centro.
`@####.##`	Reserva espaço para campo numérico com 5 casas antes do ponto decimal e duas casas após o ponto.
`@*`	Reserva espaço para informação que ocupa múltiplas linhas.
`^<<<<,`	Reserva espaço para campos preenchidos (campos multi-linha com larguras definidas).
`^>>>>,`	Cada linha deverá ser seguida por uma linha contendo a variável escalar que contém os
`^||||`	dados (mesmo repetida).
`~`	Suprime uma linha em campos preenchidos caso ela seja vazia.
`~~`	Repete o último formato (campos preenchidos) caso o texto exceda o espaço reservado previamente.
`.`	Termina a definição de um formato (se sozinho no início de uma linha)

6. Módulos

Enquanto *Perl 4* é uma linguagem totalmente orientada a procedimentos, *Perl 5* inclui todo o *Perl 4* e acrescenta uma nova estrutura que permite a construção de programas orientados a objetos.

Objetos permitem uma maior reutilização de código e a realização de tarefas complexas em menos linhas de código e de forma mais simples. Mas a programação orientada a objetos também introduz uma complexidade adicional formada por novos conceitos, termos e técnicas de programação que estão além dos nossos objetivos neste tutorial, já que nosso objetivo é oferecer recursos suficientes ao desenvolvimento de aplicações CGI típicas.

Nesta seção apresentaremos uma visão superficial dos módulos de classes – um recurso orientado a objetos do *Perl 5* – porque são úteis no desenvolvimento de aplicações em *Perl* mais sofisticadas como *gateways* de acesso a bancos de dados ou de geração de imagens, com aplicações na Web através de CGI. Vários módulos também estão disponíveis para a manipulação de dados recebidos por formulários HTML. O objetivo desta seção é, portanto, mostrar como utilizar esses módulos. Não será abordada a construção de classes e pacotes em *Perl*.

6.1. O que são módulos de classe

Módulos de classe são arquivos especiais (chamados de *pacotes*) contendo código em *Perl* e que devem ser usados para o armazenamento de funções e a construção de *objetos*. Um objeto é algo que pode armazenar dados e ocultar operações complexas, fornecendo uma lista finita de operações que podem ser realizadas sobre ele e sobre os dados que contém. As operações de um objeto são chamadas de *métodos*. Os dados que armazena, em geral, só podem ser alterados pelos métodos, que são a sua *interface pública*. Um programa orientado a objetos é, portanto, um algoritmo que, em vez de ser definido em função de uma seqüência de operações, é caracterizado pela interação entre objetos. Tais programas podem apresentar-se bem mais simples que os programas procedurais pois a maior parte da complexidade fica dentro dos objetos que não aparecem no programa principal.

Para utilizar um objeto é preciso primeiro criá-lo. Uma classe é o molde usado para isto. A *classe* contém a definição dos métodos do objeto e uma função especial chamada de *construtor*. O construtor é utilizado para criar novos objetos a partir da classe. Cada objeto tem sua própria cópia dos métodos e variáveis definidos na classe.

Em *Perl*, classes são definidas dentro de módulos. Módulos podem conter pacotes ou subpacotes que, por sua vez, contém as classes. Para usar um módulo é preciso utilizar a instrução use e o nome do(s) pacote(s) definido(s) nele. Por exemplo, para usar o módulo `ODBC.pm` que define o pacote `Win32::ODBC` é preciso importá-lo com:

```
use Win32::ODBC;
```

Os pacotes que se comportam como classes devem conter funções e construtores para a construção de objetos. No caso do módulo acima, o pacote define uma instrução `new` que funciona como construtor (é uma convenção criar construtores com este nome). A instrução requer o parâmetro `Win32::ODBC` (nome da classe – outra convenção) seguido pelo nome de uma fonte de dados ODBC. Para criar um novo objeto com esta classe, portanto, é preciso fazer:

```
$objeto = new Win32::ODBC("nome_DSN");
```

Depois que um objeto é criado, seus métodos podem ser chamados. Diferentemente dos métodos de classe (ou funções) como o construtor `new`, acima, os métodos têm que ser chamados em relação ao objeto ao qual pertencem. O operador utilizado para isto é "->":

```
$objeto->método("arg1", 99, $arg3);
$objeto->metodo2;
```

Os objetos em *Perl* são destruídos automaticamente através de um sistema de coleta de lixo, portanto, o programador não precisa se preocupar em destruir objetos criados.

6.2. Como usar módulos

É preciso conhecer quais os métodos, classes e objetos definidos em um módulo antes de usá-lo. Se o módulo faz parte da distribuição original do *Perl*, deve haver documentação disponível no diretório onde o interpretador foi instalado. Em outros casos, a documentação deve acompanhar a distribuição do módulo.

A instalação de um módulo (que não esteja disponível na distribuição *Perl*) pode consistir da simples cópia de um arquivo com extensão `.pm` para um subdiretório de `[`*dir. instalação*`]/perl/lib/`, de vários outros arquivos ou até da compilação de arquivos. É preciso consultar a documentação de cada módulo.

Na distribuição original do *Perl 5* (5.004) há vários módulos. Muitos fazem a mesma coisa de formas diferentes; uns são mais fáceis de usar, mais completos ou mais eficientes do que outros. Nas subseções a seguir, são apresentados alguns módulos úteis para o desenvolvimento de aplicações CGI.

6.3. Módulos para uso com CGI

Quase todas as distribuições do *Perl 5* (inclusive o *ActivePerl for Windows*) possuem um módulo `CGI::*` ou similar com diversas classes úteis para lidar com dados de formulários e *cookies* (`CGI::Cookie`). Esses módulos podem ser usados no lugar das bibliotecas `cgi-lib.pl` e `cookie.lib` compatíveis com o *Perl 4*. Há também módulos `HTTP::*`, `HTTP::Response::*` e `HTTP::Request::*`. Os módulos são bastante extensos e possuem muitos métodos e formas de utilização. Em vez deles, preferimos apresentar o módulo `CGI_Lite.pm`. Ele contém as funções básicas presentes na biblioteca `cgi-lib.pl` (*Perl 4*) e ainda recursos para decodificar dados no formato `multipart/form-data` (resultantes de formulários do tipo *file-upload*). Ele não faz parte da distribuição original e não tem todos os recursos dos módulos CGI nativos mas é muito mais fácil de usar. Para instalá-lo, basta copiar o arquivo `CGI-Lite.pm` para o diretório `perl/lib/` da sua instalação. CGI-Lite pode ser encontrado em qualquer repositório CPAN (*Comprehensive Perl Archive Network*) na Internet.

Exemplo de uso

O programa abaixo decodifica todos os dados recebidos pelo formulário e os imprime em um arquivo de texto devolvido ao browser:

```
use CGI_Lite;

$cgi = new CGI_Lite();
$cgi->parse_form_data();

print "Content-type: text/plain","\n\n";
```

```
$cgi->print_form_data();
```

O método `parse_form_data` decodifica os dados de entrada e os retorna em um vetor associativo. Cada valor, do par nome-valor, pode ser recuperado da forma `$vetor{'nome'}`. O programa acima usou `print_form_data` para imprimir todos os valores. O programa abaixo faz o mesmo usando o vetor `%dados`:

```
$cgi = new CGI_Lite();
%dados = $cgi->parse_form_data();

print "Content-type: text/plain","\n\n";

foreach $key (keys %dados) {
   print $key, " = ", $dados{$key}, "\n";
}
```

Exemplo com File Upload [GUND96]

Formulários que usam o elemento HTML `<input type="file">` permitem que arquivos sejam enviados ao servidor pelo cliente. Os dados recebidos precisam ser separados e decodificados. O processo é simples com `CGI-Lite`.

```
#!c:\perl\bin\perl.exe

use CGI_Lite;
$cgi = new CGI_Lite();
print "Content-type: text/plain","\n\n";
$cgi->set_directory("c:\lixo") || die "Diretorio nao existe!\n";
$cgi->set_platform("DOS");
$cgi->set_file_type("handle");

%dados = $cgi->parse_form_data();
$email = $dados{'email'};
$ARQ   = $dados{'arquivo'};

print "Eis o arquivo que você nos enviou: \n";
print "-----------------------------------\n";
if (-T $arq) {
   while ($linha = <$ARQ>) {
       print $linha;
   }
   close $ARQ;
} else {
   print "Erro: você não enviou um arquivo de texto!";
}
print "\n-----------------------------------\n";
```

O arquivo HTML com um formulário que serve de interface ao programa acima pode ser o seguinte:

```
(...) <body>
<h1>Formulário de Upload</h1>
<form action="fup.pl" method="POST" enctype="multipart/form-data">
<p>Digite seu email <input type=text name=email size=15><br>
Arquivo de texto<br><input type=file name=arquivo><br>
<button type=submit>Enviar Arquivo</button></p>
</form>
</body> (...)
```

6.4. *Módulo ODBC para* Windows

Há vários módulos *Perl* para acesso a bancos de dados no *Unix*. No *Windows*, é preciso utilizar o *ActivePerl* (`www.activestate.com`) ou versão compatível, que possui alguns módulos nativos. Para acesso via ODBC existe um módulo em `http://www.roth.net` (compatível com o *ActivePerl*). Nesse site, pode-se baixar todo o módulo ODBC para instalação ou apenas os arquivos já compilados e colocá-los nos lugares adequados (mais fácil).

Os arquivos do pacote são dois: `ODBC.PM` e `ODBC.DLL`. O primeiro deve ser copiado para `\lib\Win32` da instalação *ActivePerl* (ex: `C:\perl\lib\Win32`). O arquivo `ODBC.PLL` deve ser copiado para `\lib\Auto\Win32\ODBC\`. Depois dessa instalação, o módulo já pode ser usado.

Como usar o módulo Win32::ODBC

Seu código deverá ter a seguinte linha:

```
use Win32::ODBC;
```

Em seguida, abra uma conexão na sua fonte de dados (cria objeto `$dados`):

```
$dados = new Win32::ODBC("DataSourceName");
```

Você pode agora enviar declarações SQL à vontade através do objeto que foi obtido (veja métodos abaixo). Quando terminar de usar o banco, chame o método `Close()`:

```
$dados->Close();
```

Principais métodos

Catalog *qualifier, owner, name, type*

Recupera o catálogo do objeto ODBC atual. Retorna um vetor de quatro elementos: (*Qualificador*, *Proprietário*, *Nome*, *Tipo*). Todos os nomes de campo usam caixa-alta. Exemplo:

```
($qualifier, $owner, $name, $type) = $db->Catalog("", "", "%", "'TA-
BLE'");
```

Close

Fecha a conexão.

Data

Data *list*

Recupera os dados de um cursor previamente carregado (como resultado de uma declaração SQL). Como escalar, retorna todos os campos concatenados. Como vetor, retorna cada campo em um elemento de vetor.
Exemplo:

```
$db->Sql("SELECT f1, f2, f3 FROM foo");
$db->FetchRow();
($f1, $f2) = $db->Data("f1", "f2");
```

ou

```
$db->Sql("SELECT * FROM foo");
$db->FetchRow();
@values = $db->Data;
```

DataHash

DataHash *list*

Recupera os dados de um cursor previamente carregado (como resultado de uma declaração SQL). Retorna uma tabela associativa (nome-valor) onde o nome do campo é a chave para recuperar o valor. Exemplo:

```
$db->Sql("SELECT f1, f2, f3 FROM foo");
$db->FetchRow();
%hash = $db->DataHash("f1", "f2");
print $hash{f1};
```

ou

```
$db->Sql("SELECT * FROM foo");
$db->FetchRow();
%hash = $db->DataHash;
foreach $key (sort(keys %hash)) {
   print $key, '=', $hash{$key}, "\n";
}
```

Error

Retorna o último erro na forma de um vetor ou um *string*. *Exemplo*:

```
die $db->Error(), qq(\n);
($ErrNum, $ErrText, $ErrConn) = $db->Error();
```

FetchRow

Busca o próximo registro (linha da coluna) da última declaração SQL. Para recuperar os dados, é preciso seguir um `FetchRow` por `Data` ou `DataHash`. Retorna `undef` se não houver mais linhas para ler. *Exemplo*:

```
$db->Sql("SELECT * FROM foo");
$db->FetchRow() || die qq(Fetch error: ), $db->Error(), qq(\n);
$f1 = $db->Data("f1");
```

TableList

TableList *qualifier, owner, name, type*

Recupera uma lista de nomes de tabela da conexão ODBC atual usando `Catalog`. *Exemplo*:

```
@tables = $db->TableList;
```

CGI para acesso via ODBC

O exemplo abaixo mostra um acesso simples a um banco de dados *Access*. Para que funcione é preciso que o módulo ODBC esteja instalado (seção anterior) e que haja uma fonte de dados ODBC (*Data Source*) no sistema com o nome "mdbdados1" que aponte para a base `anuncios.mdb`[3], distribuída com esta apostila. O programa apenas lista o conteúdo da base.

```
#!c:\perl\bin\perl.exe

use Win32::ODBC;

print "Content-type: text/html\n\n";
&buscaTudo;
exit(0);

sub buscaTudo
{
      $sql = "SELECT * FROM anuncios;";
      $dsn = "mdbdados1"; # este é o nome da fonte de dados do sistema
      $bd = new Win32::ODBC($dsn);
      $bd->Sql($sql);
      print "<table border=1>";
      while ($bd->FetchRow())
      {
            @registros = $bd->Data("numero", "data", "texto", "autor");
            print "<tr valign=top>";
            print "<td>$registros[0]</td>";
            print "<td>$registros[1]</td>";
            print "<td><pre>$registros[2]</pre></td>";
            print "<td>$registros[3]</td>";
            print "</tr>";
      }
      print "</table>";
      $bd->Close();
}
```

Veja outros exemplos, com acesso completo, atualização, inserção, remoção e busca nesta base, no disquete que acompanha esta apostila.

[3] O banco de dados anuncios.mdb contém apenas uma tabela chamada 'anuncios'. As colunas são 'numero' (INT), 'data' (CHAR), 'texto' (CHAR) e 'autor' (CHAR).

6.5. Módulo gráfico GD

O módulo gráfico GD é utilizado para gerar imagens PNG (ou GIF, nas versões mais antigas). Para isto dispõe de uma biblioteca de métodos e classes para a manipulação gráfica. Para instalar o GD é preciso baixar o módulo em `www.boutell.com`, compilá-lo e instalá-lo.

No *Unix*, a instalação geralmente não apresenta problemas e é simples, pois o roteiro de instalação é bastante eficiente.

No *Windows*, a compilação exige o *GNU C Compiler*, o que não é muito comum entre os usuários desse sistema. A melhor solução, neste caso, é usar o aplicativo *Perl Package Manager* (*PPM*) do *ActivePerl*. Estando conectado à Internet, rode *PPM* e, quando o prompt "`PPM>`" aparecer, digite "`install GD`" e aguarde uns 5 minutos (enquanto ele faz download dos arquivos). Depois de instalado, rode os programas de teste. Você pode usar a biblioteca incluindo no seu programa a declaração:

```
use GD;
```

Depois, crie um objeto `Image`, `Font` ou `Polygon` e chame seus métodos através dele.

Objeto Image

`GD::`**`Image`**

É uma classe que permite a criação de um objeto através do qual pode-se manipular os dados de uma imagem e chamar métodos de desenho e transformação. O objeto é criado através do construtor (método de classe) `new`:

```
$imagem = new GD::Image(50, 50);
$imagem->rectangle(0,0,40,40, $im->colorAllocate(0,0,255));
```

Veja um exemplo de criação e exibição de uma imagem (*Manual do GD*):

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

use GD;

# create a new image
$im = new GD::Image(100,100);

# allocate some colors
$white = $im->colorAllocate(255,255,255);
$black = $im->colorAllocate(0,0,0);
$red = $im->colorAllocate(255,0,0);
$blue = $im->colorAllocate(0,0,255);

# make the background transparent and interlaced
$im->transparent($white);
$im->interlaced('true');

# put a black frame around the picture
$im->rectangle(0,0,99,99,$black);

# Draw a blue oval
$im->arc(50,50,95,75,0,360,$blue);
```

```
# And fill it with red
$im->fill(50,50,$red);

# Convert the image to PNG and print it on standard output
print $im->png;
```

Objeto Polygon

GD::**Polygon**

Classe usada para criar um polígono. Os métodos que podem ser chamados a partir do objeto permitem acrescentar e remover vértices e realizar outras transformações.

```
$poly = new GD::Polygon;
$poly->addPt(50,0);
$poly->addPt(99,99);
$poly->addPt(0,99);
```

Objeto Font

GD::**Font**

Classe usada para criar novas fontes.

Construtor newFromGif (método de classe)

Este construtor cria uma imagem (manipulável dentro de um programa) através de uma imagem GIF existente, passada através de um descritor de arquivo previamente aberto para um arquivo GIF válido. Em caso de sucesso, o construtor retorna a imagem como um objeto. Em caso de falha, retorna `undef`.

```
open (ANTAS,"duasantas.gif") || die "Impossivel abrir arquivo";
$im = newFromGif GD::Image(ANTAS) || die "Formato incorreto";
close GIF;
```

Principais métodos do objeto GD::Image

colorAllocate

GD::Image::**colorAllocate**(*r*, *g*, *b*)

Este método aloca uma cor de acordo com seus componentes de luz (vermelha, verde e azul) que podem ter valores que variam entre 0 e 255. Se uma cor não for alocada, este método retorna –1. A primeira cor alocada é utilizada como cor de fundo. *Exemplo*:

```
$white = $im->colorAllocate(255,255,255); # cor de fundo
$black = $im->colorAllocate(0,0,0);
$magenta = $im->colorAllocate(255,0,255);
```

line

GD::Image::**line**(*x1*, *y1*, *x2*, *y2*, *cor*)

Desenha uma linha de (*x1, y1*) a (*x2, y2*) na cor especificada. Pode-se usar uma cor real, previamente alocada, ou uma das cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed`. *Exemplos*:

```
$im->line(0,0,150,150,gdBrushed);
$im->line(0,150,150,0,$magenta);
$im->line(0,75,150,75, $im->colorAllocate(255,0,0));
```

rectangle

`GD::Image::`**`rectangle`**(*x1, y1, x2, y2, cor*)

Desenha um retângulo na cor especificada. Os pontos (*x1, y1*) e (*x2, y2*) correspondem aos cantos superior esquerdo e inferior direito, respectivamente. Pode-se ainda usar as cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed`. *Exemplo*:

```
$im->rectangle(10,10,100,100,$magenta);
```

filledRectangle

`GD::Image::`**`filledRectangle`**(*x1, y1, x2, y2, cor*)

Preenche um retângulo na cor especificada. Os pontos (*x1, y1*) e (*x2, y2*) correspondem aos cantos superior esquerdo e inferior direito, respectivamente. Pode-se usar uma cor real (previamente alocada) ou com o padrão definido por uma imagem (previamente carregada usando `newFromGif`) definida com o método `setTile()`. O padrão (*tile*) atual pode ser atribuído ao desenho com a "cor" reservada `gdTiled`. Exemplo:

```
open(GIF,"tijolos.gif") || die;
$tile = newFromGif GD::Image(GIF);
$myImage->setTile($tile);
$myImage->filledRectangle(10,10,150,200,gdTiled);
```

polygon

`GD::Image::`**`polygon`**(*polígono, cor*)

Desenha um polígono na cor especificada. É preciso, antes de desenhá-lo, criar o polígono usando a classe `GD::Polygon`. Um polígono deve ter pelo menos três vértices. Se o último vértice não fechar o polígono, o método o fechará automaticamente antes de desenhá-lo. Pode-se usar cores reais ou as cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed` para desenhar o polígono (não preenchido). Exemplo:

```
$poly = new GD::Polygon;
$poly->addPt(50,0);
$poly->addPt(99,99);
$poly->addPt(0,99);
$im->polygon($poly,$blue);
```

filledPolygon

`GD::Image::`**`filledPolygon`**(*polígono, cor*)

Desenha e preenche o polígono com a cor especificada. Pode-se usar as cores comuns ou a cor especial `gdTiled` (padrão baseado em imagem previamente importada). *Exemplo*:

```
$poly = new GD::Polygon;
$poly->addPt(50,0);
$poly->addPt(99,99);
$poly->addPt(0,99);
$im->filledPolygon($poly, $magenta);
```

arc

GD::Image::**arc**(*cx*, *cy*, *largura*, *altura*, *início*, *fim*, *cor*)

Desenha arcos e elipses. O centro é especificado em *cx*, *cy*) e (*largura*, *altura*) especificam a altura e largura. O *início* e o *fim* são definidos em graus de 0 a 360. Zero é a parte mais alta do elipse. Noventa (90) é a parte mais à direita. Para desenhar círculos, use início em 0, fim em 360 e valores iguais para altura e largura. Pode-se usar cores reais ou as cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed`. *Exemplo*:

```
$im->arc(100,100,50,50,0,180,$blue);
```

fill

GD::Image::**fill**(*x*, *y*, *color*)

Preenche uma região na cor especificada. Funciona como a ferramenta "balde de tinta" nos aplicativos de desenho. A pintura começa na origem (*x, y*) e pára logo que encontra um pixel de outra cor. Pode-se usar cores normais ou `gdTiled`. *Exemplo*:

```
$im->rectangle(10,10,100,100,$black);
$im->fill(50,50,$blue);
```

fillToBorder

GD::Image::**fillToBorder**(*x*, *y*, *cor_da_borda*, *cor_do_preenchimento*)

Como `fill`, este método preenche uma área com a cor especificada mas permite que se defina outra cor para a borda. A borda só pode usar uma cor normal (não pode usar as cores especiais). O preenchimento pode ser uma cor normal ou `gdTiled`. *Exemplo*:

```
$im->rectangle(10,10,100,100,$red);
$im->fillToBorder(50,50,$black,$blue); # borda será azul
```

string

GD::Image::**string**(*font*, *x*, *y*, *string*, *cor*)

Este método desenha uma *string* na posição (*x,y*) utilizando fonte e cor especificadas. Há quatro fontes diferentes que podem ser usadas: `gdSmallFont`, `gdMediumBoldFont`, `gdTinyFont` e `gdLargeFont`. *Exemplo*:

```
$im->string(gdSmallFont,2,10,"Imagem vale 1000 palavras", $magenta);
```

png (ou gif)

GD::Image::**gif**

Retorna o código (texto) da imagem em formato PNG (*Portable Network Graphics*) ou GIF (*Graphics Image Format*). Apenas um dos formatos está disponível dependendo da versão do GD utilizada. As versões mais novas suportam apenas PNG uma vez que o formato GIF agora é proprietário.

```
$imagemPNG = $im->png;
print $imagemPNG;
```

Consulte manual do GD para outros métodos e métodos dos objetos `Polygon` e `Font`.

7. Referências

1. Randal Schwartz. *Learning Perl*. O'Reilly and Associates, 1993
2. Larry Wall. *Programming Perl*. O'Reilly and Associates, 1992
3. Shishir Gundavaram. *CGI Programming for the World Wide Web*. O'Reilly and Associates, 1996
4. ActiveState. *ActivePerl 522 User´s Manual*. www.activestate.com.
5. *Perl 5.004 Manual Pages*. www.perl.org
6. *Perl 5.004 Frequently Asked Questions*. www.perl.org
7. *GD Graphics Library*. www.boutell.com
8. Roth. *Windows ODBC Module*. www.roth.net
9. Spainhour and Quercia. *WebMaster in a NutShell*. O'Reilly and Associates, 1997