

# Tutorial de Perl

Carlos Duarte  
cgd@sdf-eu.org  
Dezembro 2000

## 1. Introdução

*Perl* é uma *scripting language*. O código é escrito textualmente em ficheiros e interpretado por um programa (o interpretador *perl*). *PERL* significa “Practical Extraction and Report Language” e foi desenvolvido por Larry Wall, para executar tarefas difíceis de implementar em *shell*, mas que não justificavam a sua codificação em C. Apresenta semelhanças com C, *awk*, *sed* e *sh*.

Os seus principais pontos fortes são:

- facilidade de tratamento de texto (*text massaging*)
- alocação de memória automática
- *arrays* indexados e associativos

Desvantagens:

- o código é difícil de ler/perceber e pode tornar-se demasiado obscuro
- não tem suporte fácil para estruturas (ver “Truques e dicas”, §12, p16).

Em termos de comunidade, existe um excelente suporte, com várias centenas de módulos escritos, bem como uma diversidade de aplicações já feitas.

### 1.1. Fazer “Hello World!”

```
print "Hello World!\n";
```

A expressão de cima pode ser executada de várias formas:

(1) Linha de comando:

```
perl -e 'print "Hello World!\n";'
```

(2) *Script* file:

```
$ vi hello.pl
[edita-se o ficheiro, e insere-se]
print "Hello World!\n";

$ perl hello.pl
```

(3) Tornar o próprio ficheiro executável (apenas em Unix):

```
$ vi hello.pl
[edita-se o ficheiro, e insere-se]
#!/usr/bin/perl
print "Hello World\n";

$ chmod +x hello.pl
$ ./hello.pl
```

### 1.2. Pequeno resumo

- *perl* executa *statements*
- *statements* são separados por “;”
- comentários começam em “#”, e terminam no fim de linha
- blocos: { BLOCO }
- variáveis, tipos de dados:

\$a          escalar, pode conter strings ou valores numéricos  
@a          array: 0 ou mais escalares, indexados por inteiros

`$a[0]` primeiro escalar que `@a` contem  
`%a` hash: contem 0 ou mais escalares, indexados por strings  
`$a{'z'}` valor correspondente à chave 'z'

dados numéricos: inteiros ou reais (1234, 5.3, 2e3)

strings: qualquer sequência de texto, ou binária ('xpto', "valor de \a= \$a.")

- ciclos, *control-flow*

```
while (cond) { code }
    executa code enquanto cond for verdade. Faz primeiro o teste a cond.

do { code } while (cond)
    executa code enquanto cond for verdade . Executa primeiro code.

for (A; B; C) { code }
    equivale a: "A; while (B) { code ; C }", excepto se last/next for usado.

if (cond) { code }
    executa code (uma vez) se cond for verdade.

if (cond) { code-true } else { cond-false }
    executa code-true se cond for verdade, caso contrário executa code-false.

if (cond1) { A1 } elsif (cond2) { A2 } ...
    executa a1 se cond1 for verdade, caso contrário executa cond2 se a2 for verdade, etc...
```

operações nas condições:

```
== != >= > < <=      testes para numéricos
eq ne ge gt lt le      testes para strings
```

(igual, diferente, maior ou igual, maior, menor, menor ou igual, respectivamente)

`last` sai do ciclo mais interno

`next` salta para a próxima iteração do ciclo mais interno

`redo` salta para o início do ciclo mais interno, sem reavaliar a condição

`LABEL:`

coloca uma marca no código

`goto LABEL;`

salta para uma marca definida, saindo de qualquer ciclo em que se encontre.

- funções/rotinas.

definição:

```
sub func {
    my $arg1 = $_[0];
    my $arg2 = $_[1]; ...
    # code
    return $value;
}
```

invocação:

```
$x = func("arg1", 2);
```

## 2. Variáveis

As variáveis em *perl* são singulares ou plurais, chamando-se **escalar** às variáveis no singular, e **array** às variáveis no plural.

Por terem um símbolo que antecede o nome da variável, um escalar e um *array* podem ter o mesmo nome, não havendo colisão.

```
$foo = 1;
@foo = (1,2,3,4,5);
print "$foo\n";
print "@foo\n";
» 1
» 1 2 3 4 5
```

São automaticamente definidas e iniciadas após a sua referência no código, i.e. não necessitam de ser declaradas.

```
$a = $a + 4;          # $a = 0 + 4 = 4
print $a*3 + $b, "\n"; # print 4 * 3 + 0 = 12
» 12
```

Não existem tipos de dados (inteiros, caracteres, estruturas...), sendo que a unidade mais elementar manipulada em *perl*, é o escalar.

Apesar de não ser necessário declarar variáveis, é possível fazê-lo, usando o operador *my*. Uma variável introduzida via *my*, é apenas válida no *scope* em que foi definida (o que normalmente significa o bloco em que está inserida, tornando-a privada nesse bloco).

```
my $foo;          # global
# de facto, $foo é privada ao scope em que está definida. como neste caso
# esse scope é todo o script, para todos os efeitos, $foo é global.
$foo = 1;
print "$foo\n";   # 1
sub x {
    my $foo=5; # privada da funcao x
    print "$foo\n"; # 5
}
x();
$foo++; # incrementa $foo
print "$foo\n"; # 2
» 1
» 5
» 2
```

Para pré-declarar variáveis pode-se usar uma de duas formas (ver “variáveis que não carregam”, §12.2, p17):

```
# perl >=v5.6
our @var;      # definida algures, mas conhecida daqui para a frente

# perl >=5.005
use vars ( @var ); # semelhante (com algumas diferenças: perldoc -tf our)
```

## 2.1. Escalares

Prefixam-se por “\$” (ex: “\$a”), e podem tomar valores numéricos ou *strings* (sequências de caracteres). Representam valores singulares, i.e. cada escalar tem o valor de **um** numérico, ou de **uma** *string*.

Podem ser números:

```
$number = 4;          # notacao inteira
$number = 1.25;       # notacao real
$real = 3.4e22;       # notacao cientifica
$octal = 0377;        # notacao octal: 255 decimal
$hex = 0xff;          # notacao hexadecimal: 255 decimal
```

**Notas:**

- hexadecimal é introduzido, quando prefixado por 0x
- octal é introduzido, quando prefixado por 0
- números em *perl* são entidades reais (parte inteira e parte fraccionária). Pode-se instruir a serem usadas apenas expressões numéricas inteiras, usando o modificador “use integer;”, o que normalmente não é necessário.

Podem ser *strings*:

```
$a = "xpto";
$b = 'foo bar';
```

**Notas:**

- as *strings* podem ser introduzidas entre plicas ( ' ' ) ou entre aspas ( " " )
- entre ' ', tudo o que estiver dentro, é tratado literalmente.
- entre " ", alguns caracteres são interpretados de forma especial, incluindo os caracteres \$ e @, que provocam a expansão de escalares e *arrays*, respectivamente.

\n	fim de linha
\r	return

\t	tab
\b	backspace
\\	backslash
\"	aspa
\l	converte a próxima letra em minúscula (ex: print "\lCarlos\n";)
\L	converte em minúsculas até \E
\u	converte a próxima letra em maiúscula
\U	converte em maiúsculas até \E
\E	termina \U ou \L
\$var	expande o valor da variável \$var
@var	expande o valor de \$var[0]\$var[1]\$var[2]...

## 2.2. Arrays

Prefixam-se por “@” (ex: “@a”), contêm zero ou mais escalares, e indexam-se numericamente:

```
@a = (1, "xpto", 3.5); # dois números e uma string
@b = ();               # array vazio
@c = (1..5)            # mesmo que (1,2,3,4,5)
@copy = @c;            # copia de @c
$copy[0] = 12;         # set do primeiro elemento a 12: (12, 2,3,4,5)
```

**Notas:**

- em modo escalar ("scalar @copy", ou "\$a+@copy"), o valor de @copy é o número de elementos no array:

```
for ($i=0; $i<@a; $i++) { print $a[$i], "\n"; }
```

escreve todos os elementos de @a

- os arrays (como tudo o resto) em *perl* são *zero based*, i.e. o primeiro elemento de um array, tem index 0 (\$a[0]) e o último tem index @a-1 ou \$#a (\$a[@a-1] ou \$a[\$#a])

## 2.3. Arrays associativos (hashes)

Prefixam-se por “%” (ex: “%a”), contêm zero ou mais pares chave/valor (*key/value*), em que ambos são escalares, e indexam-se por *strings*.

Internamente, **não é mantida a ordem** dos elementos que constituem a *hash*<sup>1</sup> (ver “funções úteis para arrays associativos”, §9.5, p13).

```
%a = ("carlos", 12, "joao", 34, "xpto", 15);
$b = $a{"carlos"} # $b vale 12
$a{'joao'} = 44;   # $a{'joao'} muda de 34 para 44
```

**Notas:**

- o número de elementos de uma lista tem que ser par, para poder ser atribuída a uma *hash*.

## 2.4. Notas gerais sobre variáveis

- variáveis de diferentes tipos podem ter o mesmo nome.

```
my $a;
my @a;
my %a;
$a = "xxx";
@a = (1,2);
%a = ("xx", 3, "yy", 4);
print $a, "\n";
print $a[0], " ", $a[1], "\n";
print $a{"xx"}, " ", $a{"yy"}, "\n";
» xxx
» 1 2
» 3 4
```

---

<sup>1</sup> os elementos são distribuídos de acordo com um algoritmo de dispersão, que normalmente atribui um valor numérico inteiro às chaves.

- (2) o valor `undef` pode ser atribuído a escalares, ou membros de listas, e hashes, para designar indefinição.

Indefinição é semelhante a “vazio”, só que responde como falso ao operador `defined` enquanto vazio faz retornar verdadeiro.

```
$a = "";      # $a tem a string vazia

undef $b;     # $b está indefinido
$b = undef;   # equivalente

defined $a;   # verdade
defined $b;   # falso

$a eq $b;     # verdade (undef foi tratado como string vazia)

($a,undef,$c) = my_func(); # ignora o segundo valor retornado por my_func()
```

- (3) dependendo do contexto, as variáveis (e não só), podem ser interpretadas em modo escalar ou modo lista, com comportamentos diferentes.

```
# [lista = escalar]
@a = 5;      # escalar, visto como uma lista
@a = (5);    # equivalente

@a = "xxx";  # tb funciona com nao numericos
@a = ("xxx");

# [hash = escalar]
%a = 5;      # uma chave, com valor undef
%a = (5, undef); # equivalente

# [escalar = lista]
$n = @a;     # $n é numerico, e fica com o numero de elementos de @a

# [escalar = hash]
$n = %a;     # $n é numerico, e contem estatisticas sobre a
              # taxa de ocupacao da hash

# forcar o contexto escalar: operador "scalar"
print scalar @a, "\n"; # imprime o numero de elementos de @a
print @a, "\n";       # imprime todos os elementos de @a, separados por $, (")
print "@a\n";         # imprime todos os elementos de @a, separados por $" (" ")
```

- (4) as variáveis quando são autoiniciadas, ou introduzidas via `my`, são iniciadas a `undef` ou vazio, conforme se tratem de escalares ou não. (as variáveis não escalares têm comportamentos não intuitivos quando se lhes aplica o operador `undef`).

```
# "esvaziar" vários tipos de variaveis
undef $foo;  # ok em escalares
@foo = ();   # recomendado em listas
%foo = ();   # ... e hashes

# testar se as variaveis nao estao vazias
defined $foo and ... # pode ser perigoso fazer: "$foo and...", porque
                    # se $foo==0, falha a condicao, apesar de $foo estar definida

@foo and ...
%foo and ...
```

### 3. Operadores

Lista dos operadores de perl  
(prioridade dos operadores decresce para baixo)

left	terms and list operators (leftward)
left	->
nonassoc	++ --
right	**
right	! ~ \ and unary + and -

left	=~ !~
left	* / % x
left	+ - .
left	<< >>
nonassoc	named unary operators
nonassoc	< > <= >= lt gt le ge
nonassoc	== != <=> eq ne cmp
left	&
left	^
left	&&
left	
nonassoc	.. ...
right	?:
right	= += -= *= etc.
left	, =>
nonassoc	list operators (rightward)
right	not
left	and
left	or xor

### 3.1. Operadores de comparação

comparação	númericas	em strings
igual	==	eq
diferente	!=	ne
menor	<	lt
maior	>	gt
menor ou igual	<=	le
maior ou igual	>=	ge
comparação	<=>	cmp

#### Notas:

- em perl, falso (F) é qualquer expressão que tome o valor zero, e verdade (V) é qualquer expressão que tome um valor diferente de zero
- todos os operadores de comparação, excepto <=> e cmp, são expressões que tomam o valor 1 em caso de verdade, e 0 em caso de falsidade.

```
4 > 0      => toma o valor 1
4 > 5      => toma o valor 0
"x" gt "a" => toma o valor 1
```

- os operadores <=> e cmp, fazem uma comparação entre os dois operandos, e tomam o valor -1, 0 ou 1, conforme o primeiro operando seja menor, igual ou maior, que o segundo, respectivamente:

```
1 <=> 5      => -1 (1 é menor que 5)
"abc" cmp "aaa" => 1 ("abc" é maior que "aaa")
"a" cmp "a"   => 0
```

### 3.2. Operadores de lógica

!	negação lógica (!F=V; !V=F)
&&	e lógico (F&&F==F&&V==V&&F==F; V&&V=V)
	ou lógico (F&&F==F; F&&V==V&&F==V&&V=V)

not, and, or

operadores de lógica como em cima, mas com menos prioridade que as versões em símbolos (ver prioridades na tabela com todos os operadores)

### 3.3. Operadores de aritmética

```
$a = $b + $c;      # adição (- * / % ** subtracção multiplicação divisão
                    resto exponenciação)
```

```
$a = $string x $number;    # repete $string, $number vezes ($a="."x3 ⇒ "...")
$a = $b . $c;              # concatena $b com $c ($a = "123" . "456" ⇒ "123456")
```

### 3.4. Operadores bit a bit (bitwise)

```
& | ^ ~                # operadores lógicos, bit a bit (&: and; |: or; ^: xor; ~: neg)
<< >>                  # left, right shift ($a = $b << $d)
```

### 3.5. Atalhos

```
$a += $b;               equivale a: $a = $a + ($b)
$a ||= $b;              equivale a: $a = $a || ($b)
```

Funciona com os diversos operadores (ver tabela).

```
$a++; $a--;             pós autoincremento/autodecremento (inc: $tmp=$a; $a=$a+1; $tmp)
++$a; --$a;            pre autoincremento/autodecremento (inc: $a=$a+1; $a)
```

## 4. Blocos, condições e ciclos

Um bloco, em regra geral, é um conjunto de *statements* delimitado por chavetas.

```
{
    stmt1;
    stmt2;
    stmt3;
    # ...
}
```

Cada *statement*, (mas não um bloco), pode ser seguido por:

```
if EXPR;
    executa se EXPR tomar um valor diferente de zero

unless EXPR;
    executa se EXPR tomar um valor igual a zero

while EXPR;
    executa enquanto EXPR != 0

until EXPR;
    executa enquanto EXPR == 0

foreach LIST;
    executa para cada um dos elementos da LIST (em $_)

;    executa sempre
```

### 4.1. Condições

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

a mesma syntax funciona para unless
```

### 4.2. Ciclos

```
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK
LABEL BLOCK continue BLOCK

LABEL:
# ....
goto LABEL;
```

Notas:

- o `goto` não salta para as *labels* dos blocos
- dentro de um ciclo:

```
next [LABEL]
    salta para a proxima iteração do ciclo, reavaliando a condição
redo [LABEL]
    re-itera o ciclo, mas sem reavaliar a condição
last [LABEL]
    termina o ciclo
```

caso `LABEL` seja especificado, o ciclo afectado é o referente a `LABEL`, caso `LABEL` não seja especificado (situação mais frequente de ocorrer), o ciclo afectado é o mais interno.

Se existir bloco `continue`, será sempre executado no fim de cada iteração, mesmo se se usarem os operadores anteriores.

### 4.3. Exemplos e notas finais

As condições, ciclos e blocos, comportam-se de forma muito semelhante à linguagem C, com a grande excepção que uma condição tem que ser seguida por um bloco.

```
if ($foo > $max) { $max = $foo } # {} obrigatorias!!
```

Os modificadores podem constituir alternativas mais curtas a pequenas expressões.

```
$max = $foo if $foo > $max; # mais simples (?)
```

Um bloco toma o valor da última expressão que avaliou. Se um bloco nao puder ser avaliado como está, pode-se usar o operador `do` para se o conseguir. O `do`, quando aplicado a um bloco sem condições, agrupa o bloco como uma expressão.

```
$foo = do { my $foo = 2; $foo * 5 }; # $foo = 10 (yikes!)
```

Inclusivé, podem-se usar os modificadores normais que funcionam para expressões, na expressão formada por `do`,

```
do { ... } while (cond);
do { ... } until (cond);
do { ... } if (cond);
do { ... } unless (cond);
do { ... } for (list);
```

com a grande excepção que na forma `do { ... } while (cond);`, `cond` é avaliada **depois** do código ser executado uma vez. Em todos os outros casos, `cond` é sempre avaliada **antes** do código ser executado. Em expressões normais a `cond` é **sempre** avaliada primeiro.

```
$a=0;      $a++ while ($a != 0 && $a != 5); print $a, "\n"; # 0
$a=0; do { $a++ } while ($a != 0 && $a != 5); print $a, "\n"; # 5
» 0
» 5
```

No primeiro caso a condição foi avaliada primeiro e como `$a` valia 0, `$a++` não chegou a ser executado. No segundo caso `$a++` foi executado primeiro e a condição só se tornou verdadeira quando `$a` chegou a 5 (`$a++` executou 5 vezes).

## 5. Expressões regulares

Expressões regulares constituem uma pequena linguagem que permite fazer *match* (“encontrar”) formas regulares de texto, em texto. São constituídas pelos seguintes operadores ...

<code>\</code>	Quote the next metacharacter
<code>^</code>	Match the beginning of the line
<code>.</code>	Match any character (except newline)
<code>\$</code>	Match the end of the line (or before newline at the end)
<code> </code>	Alternation
<code>()</code>	Grouping
<code>[]</code>	Character class
<code>*</code>	Match 0 or more times



+	Match 1 or more times
?	Match 1 or 0 times
{n}	Match exactly n times
{n,}	Match at least n times
{n,m}	Match at least n but not more than m times

e aplicam-se sequencialmente. Em *perl*, são usadas explicitamente nos operadores (mais frequentes) `s///`, `m///e` `split`.

As expressões regulares são *greedy* no *match* que efectuam. I.e. se for aplicada um expressão regular que encontre todas as sequências com 1 ou mais *as*, e existirem mais que uma dessas sequências, o *match* efectua-se na mais longa.

Em *perl* o operador “?”, quando aplicado a um dos operadores quantificadores (segunda metade da tabela), tira o carácter *greedy* à expressão regular, passando esta a encontrar o primeiro *match*.

```
input -> "aa aaaa"
a+      ----
a+?     --
```

Algumas sequências são interpretadas de forma especial:

\b	Match a word boundary
\B	Match a non-(word boundary)
\w	Match a "word" character (alphanumeric plus "_")
\W	Match a non-word character
\s	Match a whitespace character
\S	Match a non-whitespace character
\d	Match a digit character
\D	Match a non-digit character

Exemplos:

```
\d*      zero ou mais dígitos: "", "1234"
\d+      um ou mais dígitos: "1234", mas não ""
\w{3}    três letras: "abc", mas não "ab,", nem "ab"
\w{2,4}  duas a quatro letras: "ab" ou "abcd", mas não "ab,", nem "abcde"
\w{,3}   três letras ou menos: "abc" ou "a", mas não "ab,", nem "abcd"
\w{3,}   três letras ou mais: "abc" ou "abcd", mas não "ab,", nem "ab"
```

## 6. Operadores especiais

`m/RE/si`

`/RE/si`

fazem *match* da expressão regular RE, em \$\_

**opção s:** trata \$\_ como uma só linha (^ faz match do início de \$\_, \$ faz match do fim)

**opção i:** faz procura em modo *case-insensitive*

`s/RE/REPLACEMENT/egis`

faz *match* da expressão regular RE em \$\_, e substitui o *match* por REPLACEMENT

**opção s:** como em m//

**opção i:** como em m//

**opção e:** avalia REPLACEMENT como uma expressão *perl*, e faz a substituição pelo seu resultado

**opção g:** faz a substituição para todos os *matches* (caso não seja especificado, faz apenas para o primeiro *match*)

`tr/SEARCHLIST/REPLACEMENTLIST/cds`

`y/SEARCHLIST/REPLACEMENTLIST/cds`

substitui em \$\_ todos os caracteres de SEARCHLIST pelos seus equivalentes em REPLACEMENTLIST.

ex: `y/aeiou/12345/`; troca todos os **a**s por **1**s, **e**s por **2**s, ...

**opção c:** complementa o sentido de SEARCHLIST (i.e. todos caracteres excepto os que lá estão)

**opção d:** apaga os caracteres presentes em SEARCHLIST e que **não** tenham equivalente em REPLACEMENTLIST.

**opção s:** squeeze — todas as sequências de dois ou mais caracteres iguais, são transformadas num só antes de se fazerem as operações

#### Notas:

- em contexto escalar:
  - `m//` retorna 1 se fez match, 0 se não
  - `s//` retorna o número de substituições feitas (0 se não fez nenhuma)
  - `tr//` retorna o número de caracteres modificados
- em contexto lista, `m//` retorna as expressões parentisadas
- se uma expressão regular for sucedida nos casos `m//` e `s///`, as seguintes variáveis ficam disponíveis para as instruções seguintes, e mesmo na REPLACEMENT string no caso `s///`:
  - `$'` parte anterior ao match (à esquerda)
  - `$&` a totalidade do match
  - `$'` parte posterior ao match (à direita)
  - `$1 $2 $3 ....` valores das expressões parentisadas na expressão regular
- pode-se usar qualquer variável com estes operadores através de: `$var =~ op///`;  
ex: `$a =~ s/^\s*//`; apaga todos os espaços iniciais em `$a`
- em `tr/LHS/RHS/` se RHS tiver menos caracteres que LHS, o último carácter de RHS é repetido virtualmente, até perfazer o mesmo número em LHS, excepto para a opção **d**, em que RHS é sempre interpretado como está. Se RHS for vazio, é feito igual a LHS, mais uma vez, excepto para a opção **d**.

## 7. Variáveis especiais

Em *perl*, existe um conjunto predefinido de variáveis que se encontram disponíveis ao programador, e que possuem diversas funcionalidades.

Algumas das mais importantes listam-se a seguir:

<code>\$_</code>	escalar usado automaticamente em muitas funções e operadores, caso nenhuma variável seja especificada
<code>\$.</code>	número da linha (record) do ficheiro corrente a ser lido
<code>\$/</code>	separador de record de entrada (default: "\n")
<code>\$_</code>	string que é impressa entre elementos de array (print @a)
<code>\$"</code>	string que é impressa entre elementos de array (print "@a")
<code>\$ </code>	<code>\$ =1</code> : para fazer autoflush; <code>\$ =0</code> : para fazer buffering dos filehandles
<code>!\$</code>	string que expressa o último erro de sistema que aconteceu
<code>?\$</code>	valor retornado (status) por comandos externos que tenham sido lançados via perl
<code>@</code>	erro do último eval
<code>\$\$</code>	PID do processo corrente
<code>\$0</code>	nome do programa que esta a ser corrido
<code>\$ARGV</code>	nome do ficheiro corrente a ser lido via <>
<code>@ARGV</code>	argumentos dados ao perl, <code>\$ARGV[0]</code> é o primeiro argumento
<code>@_</code>	os argumentos passados a uma subrotina, são recebidos neste array
<code>%ENV</code>	acesso às variáveis de ambiente, ex: <code>print \$ENV{PATH}</code> ;

`$_` Muitas funções e operadores, operam directamente nesta variável, caso não seja especificada outra de forma explícita. Noutros casos, nem sequer é possível explicitar outra variável.

```
for (@a) { print $_; }  
for my $var (@a) { print $var; } # equivalente  
  
chomp;          # opera em $_  
chomp $var;     # explicito em $var  
  
/foo/;          # vs: $var =~ /foo/  
s/xxx//;        # vs: $var =~ s/xxx//  
y/a-z/A-Z/;     # vs: $var =~ y/a-z/A-Z/  
  
# incrementa por um, o valor de cada elementos de @a  
@a = map { $_ + 1 } @a # neste caso, cada elemento de @a aparece dentro  
                        # do bloco em $_
```

`$/["\n"]`

Definição da *string* de *record*. O operador de leitura <>, lê quantidades de dados terminados por esta *string*. Por omissão, `$/` vale `"\n"`, o que significa que um *record* seja uma linha.

```

undef $/;      # lê até ao fim do ficheiro
                # (undef nunca vai ser encontrado)
$_ = <>;       # todo o input

```

\$. Contador de *records* do ficheiro corrente.

```

open F, "/etc/passwd" or die "can't open file: $!";
while (<F>) {
    print "linha no $. : $_";
}
close F;
» linha no 1 : root:*:0:0:root:/:/bin/tcsh
» linha no 2 : bin:*:1:1:bin:/bin:
» linha no 3 : daemon:*:2:2:daemon:/sbin:
» linha no 4 : adm:*:3:4:adm:/var/adm:
» linha no 5 : lp:*:4:7:lp:/var/spool/lpd:
» linha no 6 : sync:*:5:0:sync:/sbin:/bin/sync

```

\$. [""] (empty)

“output field separator”: *string* usada para separar elementos de *arrays*.

```

@a = (1,2,3);
$,=" ";
print @a, "\n"; # 123
$,="-*-"; # 1-*--2-*--3
print @a, "\n";
» 123
» 1-*--2-*--3-*--

```

\$" [" " (space)]

*string* usada para separar elementos de *arrays*, mas quando estes são interpretados entre aspas.

```

@a = ("a", "b", "c");
$" = " ";
print "@a\n"; # abc
$" = "+";
print "@a\n"; # a+b+c

# equivalentes:
$a = join($x,@a);
$" = $x; $a = "@a";
» abc
» a+b+c

```

\$| [0] Por omissão, as escritas para *file handles*, são *buffered* (i.e. armazenadas num *buffer* até terem dimensão suficiente para justificarem uma escrita para o ficheiro), excepto se o *file handle*, representar um terminal.

A variável \$" controla se o *buffering* é feito no *file handle* corrente. Se o seu valor for zero, não se faz *autoflush* (i.e. o *buffering* fica activo), se valer 1, o *autoflush* fica activo (*no buffering*).

```

# desligar o buffering no ficheiro corrente (normalmente STDOUT)
$| = 1;
print "xxxx\n" while 1;

# desligar o buffering em qq ficheiro
my $oldfh = select(MY_FILEHANDLE); $| = 1; select($oldfh);
print MY_FILEHANDLE "xxxx\n" while 1;

```

\$! Contém a mensagem de erro de última operação de sistema mal sucedida.

```

if (! open F, "no/file/foo") {
    print "nao abri o ficheiro no/file/foo, porque $!\n";
    exit(1);
}
» nao abri o ficheiro no/file/foo, porque No such file or directory

```

## 8. Subrotinas (funções)

Uma subrotina, define-se usando a palavra chave sub:

```
sub name BLOCK
```

A invocação da subrotina é feita, usando o nome da função, com argumentos opcionais entre parênteses:

```
name(args);
$a = name(args);
```

```
@r = name();
```

A função recebe argumentos em @\_ que lhe são passados como listas.

Retorna a expressão que se seguir a um return, ou a última expressão que tenha executado.

```
sub factorial {
    my $n = shift;    # shift extrai o $_[0] de @_
    if ($n <= 1) {
        return 1;    # valor de retorno explicito via return
    }
    $n * factorial($n-1) # valor implicito da ultima expressão
}
```

O uso de return é opcional, mas se existir, termina a execução da função:

```
sub factorial2 {
    my $n = shift;    # shift extrai o $_[0] de @_
    if ($n <= 1) {
        1;    # valor de retorno explicito via return
    } else {
        $n * factorial($n-1) # valor implicito da ultima expressão
    }
}
```

Exemplo:

```
sub my_sum {
    my $sum = 0;        # local var
    local $_;           # keep $_ clean on function exit
    for (@_) {          # all received args are on @_ array
        $sum += $_;
    }
    return $sum;        # return my value
}
print "sum of 1,2,3,4,5 is : ", my_sum(1,2,3,4,5), "\n";
» sum of 1,2,3,4,5 is : 15
```

## 9. Funções úteis builtin

Perl dispõe de um enorme conjunto de funções implementado na própria linguagem, que operam desde as diversas formas de dados (escalares, arrays), a interações com o sistema, *networking*, etc...

Para se aceder à mais actualizada e detalhada ajuda sobre esses *builtins*, deve executar-se:

```
perldoc perlfunc
```

e pagar a informação. Para se obter ajuda rápida sobre uma função em particular, fazer:

```
perldoc -f substr ## substituir SUBSTR pelo nome da funcao que se pretende
```

A seguir listam-se algumas das funções *builtin* usadas mais frequentemente.

### 9.1. Funções úteis em números

int(E)

parte inteira de E (int(4.12) == 4)

rand()

número aleatorio entre [0,1)

### 9.2. Funções úteis em strings

chomp

remove o fim de linha de \$\_

chomp \$var

remove o fim de linha de \$var

substr EXPR,OFFSET,LENGTH

retorna LENGTH caracteres de EXPR, a partir de OFFSET (zero based)

```
substr("foo bar", 2, 3) ⇒ "o b"
```

```
substr("foo bar", 0, 4) ⇒ "foo "
```

index STR,SUBSTR[,POSITION]

retorna o index em que SUBSTR se encontra em STR (zero based), ou -1 caso não exista

se POSITION for dado, começa a procurar a partir de substr(STR, POSITION) (i.e. ignora os primeiro POSITION caracteres de STR)

### 9.3. Funções úteis em arrays

`push @a, $a1, $a2; #...`

adiciona os escalares dados (\$a1,\$a2...) no fim do array @a

`pop @a;`

elimina o último escalar de @a, e retorna-o

`unshift @a, $a1, $a2; #...`

como push, mas coloca os novos elementos no início de @a

`shift @a;`

elimina o primeiro elemento de @a e retorna-o

`@a = reverse @a;`

retorna o conteúdo de @a revertido

`@a = sort @a;`

retorna o conteúdo de @a ordenado

```
@a=(1,2);  
push @a, 3; # @a => (1,2,3)  
$a = pop @a; # $a => 3, @a => (1,2)  
unshift @a, 0; # @a => (0,1,2)  
$a = shift @a; # $a => 0, @a => (1,2)  
@a = reverse @a; # @a => (2,1)  
@a = sort @a; # @a => (1,2)
```

`splice ARRAY,OFFSET,LENGTH`

corta LENGTH elementos de ARRAY, a partir da posição OFFSET, e retorna os elementos eliminados

`map BLOCK LIST`

mapeia todos os elementos de LIST por BLOCK, e retorna a transformação feita

- cada elemento de LIST é visto em BLOCK como \$\_  
@a = map { \$\_ + 1 } 1,2,3; # @a => (2,3,4)

`grep BLOCK LIST`

filtra todos os elementos de LIST por BLOCK, e retorna apenas os que fizeram BLOCK tomar um valor diferente de zero

- cada elemento de LIST é visto em BLOCK como \$\_  
@a = grep { \$\_ >= 2 } 1,2,3; # @a => (2,3)

### 9.4. Funções úteis em strings e arrays

`@a = split /RE/,EXPR`

divide EXPR por RE, e retorna a lista de elementos (que não contêm a RE)

- `split " "`, separa por todas as sequências de espaços

```
@a = split /;/, "1;2;a;b;"; # @a => (,1,2,a,b)
```

- notar o primeiro elemento vazio e o facto de o último ser “b”, i.e. os primeiros elementos vazios mantêm-se, enquanto os últimos elementos vazios são removidos do array retornado.

`$res = join $SEP, @ARRAY`

concentra @ARRAY, intervalado por \$SEP

```
join("-", 1,2,3) => 1-2-3
```

### 9.5. Funções úteis em arrays associativos

`@a = keys %a;`

devolve todas as chaves existentes em %a

`@b = values %a;`

devolve todos os valores existentes em %a

```
delete $a{$MY_KEY}
    apaga $MY_KEY da hash %a
each %a
    devolve todos os pares KEY,VALUE iterativamente
```

- retorna lista vazia quando não houverem mais pares

```
while (($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

## 9.6. Funções úteis em ficheiros

open F, <F>, print F, close F

Exemplo de leitura:

```
# abre ficheiro "file" para leitura
open F, "file" or die "can't open file: $!";
while (<F>) {          # lê linha a linha de F (modificavel via $/)
    chomp;
    # ...
}
close F;               # fecha ficheiro
```

Exemplo de escrita:

```
# abre "file-out" para escrita
open F, ">file-out" or die "can't open file-out: $!";
for (@a) {
    print F "$_\n"; # escreve em F
}
close F;           # fecha F
```

Outros modos de open:

```
open F, ">>file" open for append
open F, "|prog"      open pipe for write, executing prog
open F, "prog|"      open pipr for read, executing prog
```

**Nota:** fazer "perldoc -f open" para mais detalhes sobre todos os modos de open.

## 9.7. Funções úteis em directorias

opendir DIR, readdir DIR, closedir DIR

Exemplo para ler todos os nomes de ficheiros de um directorio:

```
opendir DIR, "." or die "can't opendir .: $!";
while (readdir DIR) {
    print "$_\n";
}
closedir DIR;
```

## 10. Executar PERL

perl -cw perl-script

testa, mas não executa, perl-script.

-c para fazer apenas o check

-w para dar mais warnings (avisos de potenciais erros)

perl -e 'cmd line script' args

executa directamente da linha de comando um pequeno script

perl script-file args

executa um script file

perl -p

le automaticamente todos os argumentos como ficheiros, e faz print de todas as linhas no fim do processamento de cada

perl -n

como -p, mas não faz print no fim

`perl -i[EXT]`

edita os ficheiros que receber como argumentos (de acordo com as operações no script) deixa uma copia com extensão EXT, se esta for dada a seguir a -i

## 11. Módulos

*Perl* dispõe de um método de manter um conjunto de funções e variáveis de forma independente do código de outras porções de código.

Isso consegue-se usando diferentes *namespaces*, que são introduzidas pela palavra chave `package`.

Juntamente com outro operador (`bless`), podem-se construir módulos que não interferem (de forma destrutiva, pelo menos), no código principal do utilizador, e que pode ser acedido através de referências, com alguma mágica associada (introduzida pelo `bless`).

### 11.1. Utilização de...

A utilização de módulos felizmente, é extremamente simples, e não necessita de qualquer conhecimento adicional de *perl*, do que o que é necessário para executar qualquer tarefa normal.

Os passos para aceder (e usar) um módulo, são essencialmente os seguintes:

- (1) carregar o módulo
- (2) instanciar uma variável (referência), ao código do módulo
- (3) usar os métodos, e/ou variáveis dessa variável.

Típicamente estes passos são desempenhados assim:

```
use MyModule;      # (1)

my $var = new MyModule; # (2)
#my $var = MyModule->new; # equivalente

$var->my_method();   # (3)
# ... mais metodos, mais interacao com o modulo, etc...
```

### 11.2. Programação de...

Um template muito simples para escrever um módulo, é apresentado em seguida:

```
use strict;

package XXX;

# utilizar via: my $o = new XXX;
# ou com args: my $o = new XXX(arg1, arg2);
#
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    ## inicializações em $self , com @_ (or argumentos recebidos)
    ## $self->{DATA} = ...
    ## ...
    return $self;
}

sub DESTROY {
    my $self = shift;
    ## destruction and cleanup needed on $self
}

# um método deste objecto chamado "work"
# invocado assim: my $o = new XXX; $o->work($arg1, $arg2);
#
sub work {
    my $self = shift;
    my $arg1 = shift;
    my $arg2 = shift;
}

1; ## necessário para o "use XXX" funcionar (o use vai retornar 1 verdade)
```

### 11.3. Caso de estudo: DBI

As bases de dados desempenham uma importância vital nos sistemas de informação. Em *perl*, existe um módulo genérico que permite o acesso a diferentes bases de dados, através do mesmo *interface*.

Esse *interface* comum, não isola completamente o programador das idiossincrasias de cada base de dados, mas fornece alguma familiaridade entre ambientes diferentes.

O módulo genérico de interface, chama-se DBI, e necessita internamente de um DBD:::XXX por cada base de dados diferente que se queira aceder.

No caso *Sybase*, existe o módulo DBD:::Sybase, que precisa de estar carregado no sistema, para que o DBI possa aceder a uma base de dados *Sybase*. No entanto, para o programador, apenas interessa a existência do módulo DBI (só se faz *use* desse).

Um template de acesso a uma base de dados *Sybase* é apresentado de seguida:

```
use DBI;

my $db_source = "dbi:Sybase:server=MY_SERVER;database=MY_DB";
my $db_user   = "my_user";
my $db_pass   = "my_pass";

my $dbh = DBI->connect($db_source, $db_user, $db_pass)
    or die "can not connect: $DBI::errstr.";

my $sth = $dbh->prepare(qq{
    select SourceID
    from tbl_QuoteSource
    where Description = "foo"
}) or die "can not prepare: $DBI::errstr.";
$sth->execute or die "can not execute: $DBI::errstr.";
while (my @a = $sth->fetchrow_array) {
    my $source_id = $a[0];
    # ...
}
$sth->finish;
$dbh->disconnect;
```

## 12. Truques e dicas

O *perl* é uma linguagem fácil<sup>2</sup> e com grandes potencialidades, mas algumas práticas mais descuidadas, podem levar a resultados inesperados.

Esta secção alerta para alguns desses casos.

### 12.1. Mais documentação

- A última e mais completa fonte de informação sobre *perl*, vem na própria distribuição, e pode-se aceder com o utilitário que também é instalado com o *perl*, chamado *perldoc*.

```
perldoc perl      # pagina de ajuda principal, contem apontadores
                  # para as outras
perldoc perlre    # pagina de ajuda sobre expressoes regulares
perldoc -tf open  # extrai de perlfunc a informacao especifica de open()
perldoc -q file   # encontra nas paginas de FAQ, algo sobre "file handle"
```

- Muitas vezes, os próprios *scripts perl* trazem documentação embebida no código. Essa informação pode ser extraída e formatada, com a família de utilitários distribuídos também com o *perl*, chamados *pod2xxx*.

```
pod2text /usr/bin/showtable | less
pod2man  /opt/perl/bin/sitemap.pl | nroff -man -Tascii | less -s
```

### 12.2. Cuidados a ter

- **Variáveis com as letras trocadas**

sendo as variáveis autodefinidas e iniciadas em *perl*, pode-se correr o risco de, por um erro de tipografia,

---

<sup>2</sup> sujeito a alguma discussão



obter resultados inesperados num programa

```
my $number_of_accounts = 9839;
# ...
for (my $i=0; $i<$number_accounts; $i++) { ... }
## MAL: $number_accounts é autodefinido e iniciado a zero.
## o código dentro do FOR não vai executar.

## SOLUÇÃO: usar 'use strict;' no início,
## e declarar todas as variáveis com 'my'
use strict;
my $number_of_accounts = 9839;
for (my $i=0; $i<$number_accounts; $i++) { ... }
## ERRO: o perl não executa, porque 'use strict' foi usado, e
## $number_accounts não estava previamente definido
```

- **Módulos que não carregam**

quando se escreve um módulo, ou uma colectânea de funções em geral, e se tenta carregar o ficheiro num outro *script*, muitas vezes a operação falha. Isso acontece porque os operadores `use` e `require` esperam que o seu resultado final seja verdade. Isso significa que o ficheiro que vai ser carregado, tem que terminar num valor diferente de zero. Normalmente, vê-se “1;” e terminar os módulos, para forçar a passagem do valor verdade a `require` ou `use`.

```
$ cat z1
my useful_func { return $_[0] + $_[1]; }
$ cat z2
require 'z1';
my $a = useful_func(1,2);
print $a, "\n";
$ perl z2
z1 did not return a true value at z2 line 1.

[adiciona-se 1; no fim de z1]
$ cat z1
my useful_func { return $_[0] + $_[1]; }
1;
$ perl z2
3
```

- **Variáveis que não carregam**

quando uma variável é definida num outro ficheiro, com *scope*<sup>3</sup> especificado, ter em atenção que essa variável tem que ser visível no ficheiro que a carregar, o que implica não ter *scope my* no ficheiro onde é definida, e ser declarada, mas não definida, no ficheiro que a carrega.

```
$ cat z1
%a = ( a => 1, b => 2, c => 3 );
$ cat z2
use strict;

use vars '%a'; # IMPORTANTE: %a tem de ser conhecido, mas não definido
do "z1";       # carrega o ficheiro z1

for (keys %a) {
    print "$_: ${$_}\n";
}
$ perl z2
» a: 1
» b: 2
» c: 3
```

- **Falsos valores falsos**

normalmente, avalia-se o valor de uma variável para determinar o seu estado de verdade.

```
# lê todas as linhas, no fim do input, <> retorna undef,
# que é visto como zero, que é falso, e o ciclo termina
while ($_ = <>) {
    # ...
}
```

Isto funciona, se se tiver a certeza que o operador que se usa nunca retorna os valores 0, "", "0",

---

<sup>3</sup> uma espécie de “raio de acção” da variável

0.0..., caso contrário, esses valores são vistos como falsos, e podem terminar o ciclo de forma indevida.

- **Variáveis em expressões regulares**

o valor de uma variável pode ser interpretado como uma expressão regular. Nesses casos, deve-se ter em atenção os caracteres especiais. A sua interpretação pode ser desejada ou não. Caso não seja, usar o operador `quotemeta` ou a sequência especial `\Q`.

```
# $a é interpretado como é
perl -ne '$a="cgd"; /$a/ or next;print' /etc/passwd
» cgd:x:501:501::/cgd:/bin/tcsh
# $a é quoted, antes de ser interpretado
# útil se contiver caracteres especiais
perl -ne '$a="cgd"; /\Q$a\E/ or next;print' /etc/passwd
» cgd:x:501:501::/cgd:/bin/tcsh
# equivalente à última
perl -ne '$a=quotemeta "cgd"; /\Q$a\E/ or next;print' /etc/passwd
» cgd:x:501:501::/cgd:/bin/tcsh
```

- **do { code } while cond**

na forma `"do { code } while cond"`, `code` é executado pelo menos uma vez, ao contrário dos modificadores quando aplicados a expressões normais, em que `code` é avaliado primeiro.

(ver “Exemplos e notais finais”, §4.3, p8).

- **and/or vs &&/||**

os operadores `and/or` podem ser usados, praticamente em todo o lado, para substituir os `&&/||`, sem qualquer modificação no código. Isto porque os operadores `&&/||` herdaram a prioridade que tinham em C, que, por razões históricas, no desenvolvimento da linguagem, foi estabelecida demasiado alta.

Assim, a expressão:

```
$a > $max && $max = $a;
```

falha porque é internamente associada da seguinte forma:

```
$a > ( ($max && $max) = $a )
```

o que nem sequer é válido em *perl*.

Os operadores textuais, têm a prioridade “certa” o que resolve este tipo de problemas.

```
$a > $max and $max = $a; # funciona OK
```

- **Em geral**

Em geral, ler a secção *perltrap* da documentação que é distribuída com o *perl*.

```
$ man perltrap
[ou]
$ perldoc perltrap
```

## 12.3. Alguns truques

- **<>**

o operador `<>` lê dados de um *file handle* na forma `<F>`. Se se omitir o *file handle*, todos os ficheiros nomeados em `@ARGV` são abertos sequencialmente, e `<>` retorna a próxima linha do ficheiro corrente. Desta forma, todos os ficheiros passados como argumentos ao *script perl* são tratados como um só *stream*.

```
$ perl -e 'print <>' *
[mesmo que]
$ cat *
```

**Nota:** quando se usar o operador `<>` pode ser útil determinar quando o ficheiro corrente terminou, e quando o último ficheiro terminou. Para isso, usa-se a mesma função, mas de forma diferente:

`eof` detecta fim do ficheiro corrente

`eof()` detecta fim do último ficheiro

- **undef \$/**

a variável `$/` controla o delimitador até ao qual o operador `<>` lê dados. Normalmente, vale “`\n`” o que significa que `<>` lê linha a linha. Se se fizer `undef $/`, o operador `<>` lê todo o *input* de uma só vez. Isso é muito conveniente em processamento de texto, que envolva construções, que dependam de algo que venha de trás.

```
# remove todos os comentários em C
# (muito rustico: falha nas aspas)
undef $/;
$_ = <>; # todo o input em $_
s,/\s.*?\/,/,sg;
print;
```

- **Shortcuts: expr and/or action;**

usar condicionais em pode levar a um grande número de chavetas, dado os `ifs`, `whiles`, etc... requerem chavetas a seguir às condições. Outra forma, é usar modificadores a seguir a expressões:

```
while (<>) {
    next if /^#/; # salta, se começar por # (comentários)
    # ...
}
```

A desvantagem agora, é que o sentido geral da expressão fica trocado: “salta se ...”, em vez de “se ... salta”.

Um *shortcut* genérico que se pode usar, são os operadores lógicos `and/or`, dado que cada expressão toma o valor do seu resultado final, os operadores lógicos podem ligar expressões em modo “curto-circuito” (*short-circuit*). Nesse modo, a primeira expressão que falhar, interrompe o encadeamento das que se lhe seguem.

```
while (<>) {
    /^#/ and next; # se começar por #, salta esta linha
    # saltou todas as linhas com comentarios

    my @a = split " ";
    $a[9] eq "October" or next; # se o 10o campo for "October" salta
    # apenas deixa passar as linhas que tenham "October"

    # ...
}
```

- **do{} funciona como uma expressão**

a construção `do{};` pode ser usada como uma expressão, de forma quase em tudo semelhante a uma expressão normal. A única excepção são os `do-while`, que são cobertos noutra parte deste documento.

Alguns exemplos úteis para se usar `do{}` como expressão (**não esquecer o ”;“ final!**):

```
s/.../do { my $vars; ... full perl code }/e;
cond or do { ... }; # para agrupar expressões associadas a COND
```

- **switches**

*perl* não possui *switches builtin* na linguagem. No entanto, é possível construir algumas formas alternativas:

```
# metodo 1: if-elsif
if ($a == 1) {
    # ...
} elsif ($a == 2) {
    # ...
} else {
    # "default" label
}

# metodo 2: label bloco-last
SW: {
    $a == 1 and do {
        # ...
        last SW;
    };
    $a == 2 and do {
        # ...
        last SW;
    };
    # "default" label
}
```

- **Estruturas**

*perl* não suporta estruturas de dados. Mas suporta referências, e referências são escalares. A partir desse momento, as referências podem ser assignadas livremente a escalares, que podem ser membros de *arrays* ou *hashes*.

```
\@a    referência para o array @a
[]      referência para um array anónimo
\%a     referência para o hash %a
{}      referência para uma hash anónimo

# arrays de arrays
@a1 = (1,2,3);
```

```

@a2 = (4,5,6);
@a = ( \@a1, \@a2, [7,8,9] ); # array 3x3
# $a[0]      \@a1
# $a[0]->[0]      $a1[0] = 1
# $a[0]->[2]      $a1[2] = 2
# $a[1]      \@a2
# $a[1]->[1]      $a2[1] = 5
# $a[2]      [7,8,9]
# $a[2]->[0]      7

# hashes do anterior (hashes de arrays de arrays)
%a = ( 'foo' , \@a );
# $a->{'foo'}->[1]->[2] = 6
# -----
#      \@a      ---
#      \a2      ---
#      $a2[2]

```

Qualquer referência, pode ser desreferenciada, se for envolvida entre chavetas. O uso de `->` é uma facilidade para diminuir a complexidade das expressões. A forma completa de escrever a última expressão, do exemplo anterior, é:

```

# não poupa muitos caracteres, mas é menos legível.
# ${${${$a}{'foo'}}}[1]}[2]

# %{$a} -> hash desreferenciado
# ${$a}{'foo'} -> referencia para @a , \@a
# ${${$a}{'foo'}}[1] -> $a[1], referencia para @a2, \@a2
# ${${${$a}{'foo'}}}[1]}[2] -> $a2[2], 6

```

#### • Constantes

uma forma fácil de definir funções em *perl* consegue-se através do operador `use`:

```

use constant PI => 3.14159;
use constant ADMIN => 'cgd';
use constant TMP_FILE => "/tmp/foobar.xpto.$$";

my $radius = 1;
print "area: ", PI * $radius ** 2, "\n";
print "admin user is: ", ADMIN, "\n";
print "temporary file is: ", TMP_FILE, "\n";
» area: 3.14159
» admin user is: cgd
» temporary file is: /tmp/foobar.xpto.4798

```

#### • Variáveis static

em C, é possível usar variáveis *static* no interior de funções, que formam variáveis locais, mas com “memória”. I.e. apenas são visíveis à função em que foi definida (locais), mas não perdem o seu valor entre invocações.

Em *perl*, é possível fazer uma construção semelhante, usando um bloco adicional que envolva a função, e no qual são definidas as variáveis:

```

{
    my $counter=0;
    sub counter {
        return ++$counter;
    }
}

print counter(), "\n";
print counter(), "\n";
print counter(), "\n";
» 1
» 2
» 3

```

#### • facilidades de *s///* e *m//*

os operadores *s///* e *m//* aliados ao uso de expressões regulares, à forma como modificam as algumas variáveis (*\$n* e *\$&*) e o facto de retornarem valores sobre o sucesso da sua operação, constituem um excelente e conciso método para se obterem resultados, embora nem sempre de forma muito legível.

```

# imprime todos os hrefs existentes numa pagina HTML
undef $/;
$_ = <>;
my $n=0;

```

```
while (s/^.*?<a href="(.*?)\1>/si) {
    printf "url %d: %s\n", ++$n, $2;
}
```

## 12.4. One liners

Em *perl*, dada a sua facilidade no tratamento de sequências de texto, é fácil emular alguns comandos típicos de **Unix**, em poucas linhas, tal como executar tarefas úteis diversas.

Alguns exemplos apresentam-se em seguida.

- **Concatenar ficheiros no output**

```
$ cat *
$ perl -e 'print <>' *
```

- **Contar caracteres**

```
$ wc -c *
$ perl -e 'undef $/;while(<>){printf "%7d %s\n",length,$ARGV; close ARGV}' *
```

- **Contar palavras**

```
$ wc -w *
$ perl -e 'undef $/; while (<>) {
    $n=0; $n++ while s/^\\s*\\S+//s;
    printf "%7d %s\n", $n,$ARGV; close ARGV}' *
```

```
[modificado para dar um significado mais exacto de palavra: \\s,\\S    \\w,\\W ]
$ perl -e 'undef $/; while (<>) {
    $n=0; $n++ while s/^\\w*\\W+//s;
    printf "%7d %s\n", $n,$ARGV; close ARGV}' *
```

- **Contar linhas**

```
$ wc -l *
$ perl -e 'undef $/;while(<>){printf "%7d %s\n",tr/\\n//,$ARGV; close ARGV}' *
```

- **Ordenar**

```
$ sort *
$ perl -e 'print sort {chomp($aa=$a);chomp($bb=$b); $aa cmp $bb} <>'
```

- **Seleccionar as 10 primeiras linhas**

```
$ head file
$ perl -e 'print splice(@{<>},0,10)' file
$ perl -e 'undef $/;$_=<>;s/^\\(.*?\\n){,10}.*/$1/s' file
$ perl -e 'undef $/;$_=<>;/^\\(\\[\\^\\n]*\\n){1,10})/ and print $1' file
```

- **Seleccionar as 10 últimas linhas**

```
$ tail file
$ perl -e 'print splice(@{<>},-10)' file
```

- **Numerar linhas**

```
$ cat -n *
$ perl -ne 'printf "%6d\\t%s",$.,$_' *
```

- **Squeeze input lines**

```
$ cat -s *
$ perl -e 'undef $/;$_=<>;s/\\n\\n+/\\n\\n/g;s/^\\n*//s;print' *
```

- **Mostrar apenas linhas diferentes**

(desde que estejam ordenadas)

```
$ uniq *
$ perl -ne '$o eq $_ and next; print $o=$_'
```

- **Não mostrar linhas que existam repetidas**

```
$ uniq -u *
$ perl -ne 'BEGIN{$n=1}
    if($o eq $_){$n++;next}else{$n||print $o;$o = $_;$n = 0}
    END{$n||print $o}'
```

- **Mostrar apenas as linhas duplicadas**

```
$ uniq -d
$ perl -ne 'if($o eq $_){$n++;next}else{$n&&print $o;$o = $_;$n = 0}
    END{$n&&print $o}' *
```

- **Mover todos os ficheiros para letra minúscula**

```
# upper to lower: AKAKAK -> akakak
$ ls|perl -ne 'chomp;rename $_,lc $_ if $_ ne lc $_'
```

- ```
# lower to upper: abcabC -> ABCABC
$ ls|perl -ne 'chomp;$n=uc $_;$n ne $_ and rename $_,$n'
```
- **Mover todas as imagens para uma directoria**

```
# move todos os jpeg,gif,png para a subdiretório IMGS do directorio corrente
$ file *|perl -ne '/^(.*?):.*?(JPEG|GIF|PNG)/ and rename $1, "IMGS/$1"'
```
  - **Converter fins de linha de DOS para Unix (CRLF → LF)**

```
$ perl -p -i -e 's/\r$//' file
```
  - **Centrar linhas**

```
$ perl -pe 'print " " x int((72-length)/2)' file
```
  - **Misturar linhas**

```
$ perl -e '@a=<>;print splice(@a,rand(@a),1) while @a' file
```

## 13. Índice

|                            |        |                             |           |                           |    |
|----------------------------|--------|-----------------------------|-----------|---------------------------|----|
| \$!                        | 10, 11 | builtin map                 | 13        | pod2text                  | 16 |
| \$"                        | 10, 11 | builtin open                | 14        | quotemeta                 | 18 |
| \$\$                       | 10     | builtin opendir             | 14        | redo                      | 8  |
| \$,                        | 10, 11 | builtin pop                 | 13        | <b>referência</b>         | 20 |
| \$.                        | 10, 11 | builtin print               | 14        | require                   | 17 |
| \$/                        | 10     | builtin push                | 13        | s///                      | 9  |
| \$0                        | 10     | builtin rand                | 12        | <i>scope</i>              | 3  |
| \$?                        | 10     | builtin readdir             | 14        | <i>scripting language</i> | 1  |
| \$@                        | 10     | builtin reverse             | 13        | sort                      | 21 |
| \$ARGV                     | 10     | builtin shift               | 13        | <i>static</i>             | 20 |
| \$                         | 3, 8   | builtin sort                | 13        | <i>strings</i>            | 3  |
| \$__                       | 10     | builtin splice              | 13        | sub                       | 11 |
| \$                         | 10, 11 | builtin split               | 13        | <i>switch</i>             | 19 |
| %ENV                       | 10     | builtin substr              | 12        | tail                      | 21 |
| %                          | 4      | builtin unshift             | 13        | tr///                     | 9  |
| &&                         | 18     | builtin values              | 13        | undef                     | 5  |
| ()                         | 8      | <b>cat -n</b>               | 21        | uniq -d                   | 21 |
| *                          | 8      | <b>cat -s</b>               | 21        | uniq -u                   | 21 |
| +                          | 8      | cat                         | 21        | uniq                      | 21 |
| ->                         | 20     | constant                    | 20        | unless                    | 7  |
| .                          | 8      | <b>contantes</b>            | 20        | until                     | 7  |
| //                         | 9      | continue                    | 8         | <i>upper to lower</i>     | 21 |
| <>                         | 14, 18 | do                          | 17        | use strict                | 17 |
| ?                          | 8      | do{}while                   | 8, 18, 19 | use                       | 17 |
| @ARGV                      | 10     | eof, eof()                  | 18        | <b>valores numéricos</b>  | 3  |
| @                          | 4      | <b>escalar vazio</b>        | 5         | <b>verdadeiro</b>         | 6  |
| @_                         | 10     | <b>escalar</b>              | 2, 3      | wc -c                     | 21 |
| <i>CRLF para LF</i>        | 22     | <i>escape sequences</i>     | 3         | wc -l                     | 21 |
| DBI                        | 16     | <b>estruturas</b>           | 19        | wc -w                     | 21 |
| <i>Hello World</i>         | 1      | <b>expressões regulares</b> | 8         | while                     | 7  |
| <i>I/O</i>                 | 14     | <b>falso</b>                | 6, 17     | y///                      | 9  |
| RE \x                      | 9      | for                         | 7         | <i>zero based</i>         | 4  |
| <i>Sybase</i>              | 16     | foreach                     | 7         | {n,m}                     | 8  |
| []                         | 8      | <b>funções</b>              | 11        | {n, }                     | 8  |
| \"                         | 3      | goto                        | 7         | {n}                       | 8  |
| \E                         | 3      | <i>greedy</i>               | 9         |                           | 8  |
| \L                         | 3      | <b>hash vazio</b>           | 5         |                           | 18 |
| \Q                         | 18     | <i>hashes</i>               | 4         |                           |    |
| \U                         | 3      | head                        | 21        |                           |    |
| \\                         | 3      | <b>hexadecimal</b>          | 3         |                           |    |
| \                          | 8      | if                          | 7         |                           |    |
| \b                         | 3      | <i>key/value</i>            | 4         |                           |    |
| \l                         | 3      | <i>label</i>                | 7         |                           |    |
| \n                         | 3      | last                        | 8         |                           |    |
| \r                         | 3      | <b>linha de comando</b>     | 1         |                           |    |
| \t                         | 3      | <b>lista vazia</b>          | 5         |                           |    |
| \u                         | 3      | m//                         | 9         |                           |    |
| ^                          | 8      | <b>modificadores</b>        | 7         |                           |    |
| and                        | 18     | <b>modo escalar</b>         | 4, 5      |                           |    |
| <b>arrays associativos</b> | 4      | my                          | 3, 17     |                           |    |
| <i>arrays</i>              | 4      | <i>namespace</i>            | 15        |                           |    |
| bless                      | 15     | next                        | 8         |                           |    |
| <b>bloco</b>               | 7      | <i>non greedy</i>           | 9         |                           |    |
| builtin chomp              | 12     | <b>octal</b>                | 3         |                           |    |
| builtin close              | 14     | or                          | 18        |                           |    |
| builtin closedir           | 14     | package                     | 15        |                           |    |
| builtin delete             | 14     | perl -cw                    | 14        |                           |    |
| builtin each               | 14     | perl -e                     | 14        |                           |    |
| builtin grep               | 13     | perl -i                     | 15        |                           |    |
| builtin index              | 12     | perl -n                     | 14        |                           |    |
| builtin int                | 12     | perl -p                     | 14        |                           |    |
| builtin join               | 13     | perldoc                     | 12, 16    |                           |    |
| builtin keys               | 13     | pod2man                     | 16        |                           |    |





## Table of Contents

|                                                |    |
|------------------------------------------------|----|
| 1. Introdução .....                            | 1  |
| 1.1 Fazer “Hello World!” .....                 | 1  |
| 1.2 Pequeno resumo .....                       | 1  |
| 2. Variáveis .....                             | 2  |
| 2.1 Escalares .....                            | 3  |
| 2.2 Arrays .....                               | 4  |
| 2.3 Arrays associativos (hashes) .....         | 4  |
| 2.4 Notas gerais sobre variáveis .....         | 4  |
| 3. Operadores .....                            | 5  |
| 3.1 Operadores de comparação .....             | 6  |
| 3.2 Operadores de lógica .....                 | 6  |
| 3.3 Operadores de aritmética .....             | 6  |
| 3.4 Operadores bit a bit (bitwise) .....       | 7  |
| 3.5 Atalhos .....                              | 7  |
| 4. Blocos, condições e ciclos .....            | 7  |
| 4.1 Condições .....                            | 7  |
| 4.2 Ciclos .....                               | 7  |
| 4.3 Exemplos e notas finais .....              | 8  |
| 5. Expressões regulares .....                  | 8  |
| 6. Operadores especiais .....                  | 9  |
| 7. Variáveis especiais .....                   | 10 |
| 8. Subrotinas (funções) .....                  | 11 |
| 9. Funções úteis builtin .....                 | 12 |
| 9.1 Funções úteis em números .....             | 12 |
| 9.2 Funções úteis em strings .....             | 12 |
| 9.3 Funções úteis em arrays .....              | 13 |
| 9.4 Funções úteis em strings e arrays .....    | 13 |
| 9.5 Funções úteis em arrays associativos ..... | 13 |
| 9.6 Funções úteis em ficheiros .....           | 14 |
| 9.7 Funções úteis em directorias .....         | 14 |
| 10. Executar PERL .....                        | 14 |
| 11. Módulos .....                              | 15 |
| 11.1 Utilização de... .....                    | 15 |
| 11.2 Programação de... .....                   | 15 |
| 11.3 Caso de estudo: DBI .....                 | 16 |
| 12. Truques e dicas .....                      | 16 |
| 12.1 Mais documentação .....                   | 16 |
| 12.2 Cuidados a ter .....                      | 16 |
| 12.3 Alguns truques .....                      | 18 |
| 12.4 One liners .....                          | 21 |
| 13. Índice .....                               | 23 |