# Manual Conciso de `Perl`

José A. Soares Augusto
*Prof. Auxiliar*, Sec. Electrónica, DEEC, **Instituto Superior Técnico**, Lisboa, PORTUGAL
*Investigador*, **INESC-ID**, Lisboa, PORTUGAL

Versão 2.0.0, Outubro de 2001

## Conteúdo

**1 Introdução** **3**

1.1 Instalação e invocação do *Perl* . . . . . 5

1.2 A quem se dedica este Manual? . . . . . 7

**2 Breve digressão no *Perl*** **8**

2.1 Invocação do interpretador . . . . . 8

2.2 Particularidades do *Perl* . . . . . 9

2.3 'Manpages' do Perl . . . . . 12

**3 Variáveis** **14**

3.1 Escalares . . . . . 14

3.1.1 Notas sobre referências . . . . . 15

3.1.2 'Strings', listas e expressões regulares. Interpolação . . . . . 17

3.1.3 Operadores aplicáveis a escalares . . . . . 19

3.1.4 Funções matemáticas . . . . . 21

3.2 'Filehandles' (referências para ficheiros) . . . . . 22

3.2.1 Operadores de teste a ficheiros . . . . . 24

3.2.2 Tópicos mais avançados na utilização de `<...>` . . . . . 24

3.2.3 Outras funções de acesso a ficheiros . . . . . 25

3.3 Dirhandles (referências para directorias) . . . . . 25

3.4 'Arrays' (ou vectores ou listas) . . . . . 26

3.4.1 Conversão entre 'arrays' e 'strings' . . . . . 29

3.5 Contexto escalar e contexto de lista . . . . . 30

3.6 Programação funcional: `grep` e `map` . . . . . 31

3.7 'Hashes' (dicionários) . . . . . 32

3.8 Variáveis especiais . . . . . 33

**4 Referências: princípios básicos** **39**

4.1 Criação de referências . . . . . 39

4.2 Utilização de referências . . . . . 41

4.3 Referências Simbólicas . . . . . 43

**5 Processamento condicional e iterativo** **45**

5.1 Selecção e processamento condicional . . . . . 45

5.2 Iteração . . . . . 46

5.3 Modificadores . . . . . 49

5.4 Goto . . . . . 50

**6 Subprogramas, subrotinas ou funções** **51**

6.1 Variáveis locais . . . . . 52

6.2 Passagem de argumentos para subrotinas . . . . . 53

**7 Execução de código em 'run-time': `eval` e `require`** **56**

**8 Expressões regulares (*regexes*)** **58**

8.1 Sintaxe ampliada ('extended') de expressões regulares . . . . . 61

8.2 Regras aplicáveis em expressões regulares . . . . . 61

8.3 Pesquisa, substituição e transliteração . . . . . 61

**9 Manipulação de arrays de arrays em Perl** **66**

9.1 Declaração e acesso a arrays de arrays (AoAs) . . . . . 66

9.2 Criação dinâmica de AoAs . . . . . 67

9.3 Acesso e impressão . . . . . 69

9.4 Fatias . . . . . 70

9.5 Arrays de hashes, hashes de hashes, etc... . . . . 71

**10 Outras Funções em Perl** **73**

**11 Conclusão** **76**

# 1 Introdução

O *Perl* é a mais conhecida e utilizada linguagem de 'scripting' (LdS) na actualidade. Pode ser instalada em quase todas as plataformas e sistemas operativos (SOs) existentes – em mais do que o Java! – e é muito versátil. Além do mais, é distribuída como 'freeware' e há milhares de 'sites' na Internet com conteúdo relacionado com o *Perl* incluindo, por exemplo, milhares de 'scripts' CGI para utilizar em páginas no 'World Wide Web (WWW)'. Também existem disponíveis centenas de módulos (ou bibliotecas) destinados às mais variadas tarefas de programação (e.g., funções matemáticas, compressão/descompressão de arquivos, cifragem de informação, geração/interpretação de ficheiros HTML e XML, comunicação, acesso a bases de dados, etc...)

O site oficial do *Perl* é `http://www.perl.com` e contém muitos links para outras fontes de informação úteis. Existe um arquivo, denominado CPAN ('Comprehensive Perl Archive Network'), que tem espelhos ('mirrors') um pouco por por todo o mundo (incluindo um muito lento no IST!), de onde podem ser descarregados os principais módulos de extensão disponíveis para a linguagem. Os directórios da área da programação dos motores de pesquisa (e.g., do `Google`) contêm centenas de links para sites relacionados com o *Perl*.

O *Perl* partilha as qualidades, defeitos, vantagens e desvantagens de outras LdS, das quais destacamos o `python`, o `ruby`, o `Tcl/Tk` e, em menor grau por ser muito menos versátil, o `awk` (ou a sua versão GNU, o `gawk`). O *Perl* é uma linguagem semi-interpretada/semi-compilada, embora à luz dos tradicionais conceitos de compilação/interpretação seja 'aparentemente' interpretada, o que permite 'programar-executar' consecutivamente, pois não existem as fases intermédias (e lentas) de compilação e de ligação ('link') dos programas: após escrever um 'script' em *Perl* com um simples editor de texto (e.g. o básico *Notepad*) pode-se, imediatamente, executá-lo e observar o resultado ou a mensagem de erro `8-)`.

Há outras linguagens que usam o *Perl* como modelo e que partilham com ele alguma da sintaxe. A linguagem `ruby` (`www.ruby-lang.org`) é um bom exemplo, sendo a sua característica mais notória o facto de ser inerentemente uma liguagem orientada para objectos, implementando herança, polimorfismo, sobrecarga ('overload') de operadores, etc: o `ruby` pode ser descrito sumariamente como um `perl` orientado para objectos. Outra linguagem aparentada com o Perl é o `PHP,` largamente utilizado em servidores Web em tarefas de 'server-side-scripting' associado, por exemplo, a bases de dados como seja a `mySQL.`

O *Perl* (como as restantes LdSs) permite uma eficiência de programação, comparativamente a linguagens compiladas, estimada entre 2 e 10 vezes maior, quer em tempo de desenvolvimento, quer em número de linhas de código escritas. O autor, que utiliza regularmente há largos anos o *Perl* (e também o `awk` e o `python`), e que já escreveu uns largos milhares de linhas de código nestas linguagens, estima que a eficiência de programação é melhorada aproximadamente de um factor de 5, comparativamente ao C.

Refira-se que o `python` é a segunda LdS mais divulgada, a seguir ao *Perl*. Tem a seu favor o facto de ter evoluído de uma forma mais 'organizada' que o *Perl* e de ter, possivelmente, uma documentação de melhor qualidade. Além disso, é mais fácil ler e entender scripts em `python` do que em *Perl*, o que implica que é mais fácil manter código escrito em `python`. Porém, o facto de o *Perl* ter sido a LdS que acompanhou o crescimento da Web e de ter um leque de comandos mais sucintos e poderosos (embora crípticos...), leva a que o

número de seguidores do *Perl* ainda exceda largamente os apologistas do `python`. Não se esqueça que há quem defenda com mais fundamentalismo a sua linguagem de programação preferida do que a religião que pratica ou, mesmo, do que o clube de futebol de que é adepto `B-)`.

Encaradas até há pouco tempo como parentes pobres da 'programação' séria, talvez por serem desenvolvidas por programadores voluntários e não terem companhias de renome a suportá-las, actualmente as LdSs são respeitadas e adoptadas por grandes organizações. Por um lado, o seu desenvolvimento e evolução já estabilizaram e, por outro, apesar de serem quase sempre software distribuído em regime livre ('freeware'), começou a haver suporte comercial por parte de companhias que efectuam formação, prestam apoio técnico e instalam sistemas 'à la carte' optimizados para determinadas funções. Por exemplo, a companhia *ActiveState* que mantém a distribuição oficial do *Perl* para as várias versões do Microsoft Windows e, também, do `python` e do `Tcl/Tk`, é fortemente financiada pela Microsoft e emprega um largo número de programadores que foram muito importantes na fase de desenvolvimento e divulgação das LdSs.

O mote do *Perl* é *"There's more than one way to do it"(TMTOWTDI).* De facto, para realizar uma tarefa em *Perl* há sempre muitas opções! Cada programador deverá, em princípio, escolher a melhor (relativamente ao parâmetro de desempenho pretendido) das opções que domina `8-)`. Por exemplo, embora se possa realizar programação orientada para objectos (OOP) em *Perl*, virtualmente 100% da funcionalidade da linguagem está disponível mesmo sem enveredar por aquela metodologia de programação.

Neste manual expõe-se o núcleo da linguagem de programação *Perl*, tentando realçar as vantagens da linguagem no que respeita à eficiência e à rapidez da escrita de programas. O *Perl* é capaz de agradar a uma vasta gama de programadores, pois justifica o mote TMTOWTDI: mesmo programadores inexperientes podem retirar, rapidamente, após dominar as construções básicas da linguagem, benefícios da poderosa infraestrutura de processamento que o *Perl* disponibiliza.

Para além de explicar os fundamentos do *Perl* e alguns dos seus aspectos peculiares (e.g., a diferente interpretação de expressões em contexto escalar e em contexto de lista) frequentemente recorrendo a blocos de código mínimos, expomos sucintamente as principais funções e operadores do *Perl*. Na abordagem de alguns tópicos, segue-se de perto as exposições dos manuais. A profundidade da exposição varia: por exemplo, a relação entre referências e estruturas de dados complexas e as expressões regulares, são alguns dos tópicos tratados com detalhe. Em contrapartida, o modelo de orientação para objectos que é possível utilizar no *Perl* é bastante complexo (de certa maneira, foi 'enxertado' na linguagem) e é dispensável numa primeira abordagem.

Uma grande parte da popularidade do *Perl*, do `python` e do `ruby`,deve-se à existência de módulos (ou 'bibliotecas') grátis que permitem realizar tarefas específicas (funções matemáticas especiais, 'parsing' de HTML e de XML, comunicações, acesso a bases de dados, etc...). Estes módulos são normalmente mantidos por voluntários, mas quer a correcção de 'bugs', quer o seu melhoramento, são normalmente muito rápidos, pelo que muita gente os usa. Neste documento é impossível referi-los a todos, remetendo-se os interessados em realizar tarefas específicas à consulta do CPAN, no caso do *Perl*, ou dos sites `www.python.org` e `www.ruby-lang.org`, no caso do `python` e do `ruby`, que listam milhares de módulos e sub-módulos destinados às mais diversas aplicações.

## 1.1 Instalação e invocação do *Perl*

Vai discutir-se, brevemente, a instalação e a invocação do *Perl* em ambientes Windows (Windows95/98/NT/2000/Millenium) e Unix (ou as suas variantes: Linux, Solaris, Aix, Hp-Ux,...). Perdoem-me os fãs do Mac (principalmente estes...) mas nunca utilizei *Perl* em ambiente Macintosh. Fiquem, porém, a saber que existe imensa informação sobre a utilização do *Perl* em Mac, incluindo obviamente binários para instalação. O tamanho dos binários da versão actual (*Perl* 5.8.0) ronda os 11,5 megabytes, variando um pouco consoante o sistema operativo (SO).

A instalação do *Perl* em Unix (ou em SOs semelhantes, como o Linux...) é feita numa de três formas: ou existe uma versão do *Perl* incluída no sistema operativo (o que é verdade em mais de 90% dos casos...), ou existem binários disponíveis para a arquitectura, ou é compilado o código fonte ('code source') com o `gcc` ou outro compilador de C. Normalmente, não há grandes problemas neste processo, tendo-se apenas de seleccionar os módulos que se pretende compilar. Estes módulos são, em geral, escritos numa mistura de *Perl* e de C.

Em DOS/Windows, até há 3 ou 4 anos era difícil utilizar cabalmente o *Perl*. Actualmente, este já não é o caso. A melhor opção disponível consiste em utilizar a distribuição oficial do *Perl* para Windows, mantida pela companhia *ActiveState* (`http://www.activestate.com`). Esta distribuição utiliza o formato de instalação `.msi` da *Microsoft* (para Windows95/98 deverá ser feita uma instalação prévia do instalador da *Microsoft* que se encontra disponível no site da *ActiveState* ou, obviamente, no site da *Microsoft*). Diga-se que a *ActiveState* é financiada pela *Microsoft* `B-)` e que o *Perl* e o `python` vão ser incluídos na próxima versão do *Visual-Studio* da *Microsoft*. Aquela distribuição oficial inclui módulos específicos para Windows que permitem aceder a aplicações da *Microsoft* (Word, Excel, Access,...) e de outras companhias de software a partir do *Perl*.

Existe também uma distribuição de binários para Windows, compilada com *VisualC*, denominada *indigoperl*. Esta distribuição é mantida pela companhia *Indigostar* (`http://www.indigostar.com`) e tem a particularidade de incluir o servidor Apache compilado com vários módulos, de onde se destaca o `modperl` que permite utilizar o *Perl* como linguagem de 'server-side-scripting' (o interpretador é executado concorrentemente com o servidor, o que acelera a execução de scripts em *Perl* embebidos em páginas de `HTML`).

Ainda em Windows, poderá ser utilizado o *VisualC* (5 ou 6) para compilar as 'sources' do *Perl* ou, em alternativa, o 'sistema' *Cygwin* (um 'port' do ambiente Unix para Windows que inclui ferramentas como o `make` e o `gcc`). Existem binários do *Perl* já pré-compilados com o *Cygwin*. Recentemente, a *Siemens* disponibilizou ainda outra distribuição compilada do *Perl*. O CPAN divulga os links para estes e outros binários.

Vai-se enumerar as vantagens e as desvantagens associadas às LdS. As **desvantagens das linguagens de 'scripting'**, relativamente às linguagens compiladas são:

- execução de 'scripts' mais lenta do que programas compilados (dependendo das tarefas, entre 2 e 20 vezes)
- as LdSs têm sofrido uma evolução muito rápida, por vezes com ligeiras incompatibilidades entre versões (embora, seja dita a verdade, se tente sempre

preservar a retro-compatibilidade)

- muitas LdSs não exigem a declaração das variáveis, o que permite programar rapidamente; porém, os interpretadores não alertam para erros que as linguagens compiladas, onde a declaração de variáveis é obrigatória, alertam
- devido às poderosas instruções disponíveis nas LdS, a leitura e compreensão dos programas é por vezes difícil (o *Perl* é o principal 'suspeito' neste ítem...), o que exige do programador a disciplina necessária à escrita de boa documentação.

Por outro lado, as **vantagens das linguagens de 'scripting'** são:

- ser à borla!
- permitir desenvolver **rapidamente** programas, o que 'enche de orgulho' os programadores. A disponibilidade de uma grande quantidade de módulos e bibliotecas, organizados, sujeitos a manutenção constante e bem documentados, ajuda à sua divulgação
- dispôr de **instruções sintéticas** e **poderosas** para processar texto (procurar, substituir, ordenar, separar,...) e suportam sintaxes complexas de expressões regulares
- quase todas suportam OOP, sendo as mais evoluídas neste domínio o `ruby` e o `python.` Muitas incluem poderosas instruções de programação funcional (filtros, iteradores,...)
- estão disponíveis para muitas plataformas, sendo a **compatibilidade** entre scripts escritos em várias plataformas quase de 100% (descontando, obviamente, as tarefas específicas a cada SO)
- permitem programar **extensões** em `C` e algumas (o `python`, por exemplo) têm interpretadores escritos em `Java` e a possibilidade de utilizar a grande quantidade de classes de `Java` disponíveis
- a **documentação** disponível para as linguagens é **excelente.** Neste ponto destacam-se o `python` e o *Perl*. A documentação do `awk` é boa, mas a linguagem é relativamente pouco potente. O `ruby` é uma linguagem que se desenvolveu durante muito tempo quase exclusivamente no Japão, pelo que a sua documentação em inglês era apenas razoável: se souber ler Japonês parabéns `B-)`. No entanto, foram publicados recentemente alguns excelentes livros de `ruby`, em Inglês. Um desses livros está disponível 'on-line'.
- fazem **automaticamente** a **gestão** de **memória** dinâmica: reservam e libertam memória quando as variáveis são criadas ou eliminadas. Lembre-se que a gestão de memória é uma das grandes causas de 'bugs' em C.

Segundo o criador do *Perl*, Larry Wall, esta linguagem visa suprir os três principais defeitos do programador típico: "laziness, impatience, and hubris"(i.e., preguiça, impaciência e 'mania'!). Leia a Bíblia do *Perl*, o livro de Larry Wall, Tom Christiansen e Randall Schwartz, *Programming Perl*, da O'Reilly & Associates, para descobrir porquê!

## 1.2 A quem se dedica este Manual?

Aos alunos de SVTCE, obviamente, `8-)`. O *Perl* é excelente para processar o tipo de ficheiros associado a ferramentas de simulação e de teste de circuitos eléctricos, que são habitualmente simples ficheiros de texto.

Não se presume que o leitor vai aprender os fundamentos de programação com este manual. No entanto, quem souber programar razoavelmente numa outra linguagem decerto não terá dificuldade em seguir, até ao fim, este texto. Nas partes menos claras ou omissas (o que é feito propositadamente, devido à escassez de tempo), sugere-se a consulta de um bom livro de *Perl* ou dos manuais que são instalados juntamente com os ficheiros executáveis da linguagem.

# 2 Breve digressão no *Perl*

## 2.1 Invocação do interpretador

A execução em Unix de um programa em *Perl* denominado, por exemplo, `script.pl` é feita da seguinte forma

```
unix> perl script.pl arg1 arg2 ...
```

onde `unix>` representa a linha de escrita de comandos (ou consola de comandos) em Unix e `arg1, arg2,...` são os (eventuais) argumentos passados para o programa. O executável do *Perl* deverá estar na directoria `/bin`, `/usr/bin`, ou noutra em que o SO a encontre (i.e., no 'path').

Alternativamente, se `script.pl` fôr tornado executável com o comando

```
unix> chmod +x script.pl
```

e a linha inicial do programa `script.pl` fôr

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

(supondo que o *Perl* se encontra na directoria `/usr/bin`) ele é executado com

```
unix> script.pl [arg1 arg2 ...]
```

Em DOS/Windows95/98 tem de se fazer obrigatoriamente

```
dos> perl script.pl arg1 arg2 ...
```

embora em WindowsNT/2000 se possa tornar executáveis scripts em *Perl* através do registo (pelo administrador do sistema) do sufixo `.pl` como sendo executado com o progama `perl.exe` (não sei os detalhes todos, pois não uso WindowsNT/2000).

Em algumas distribuições para Windows95/98 existe um programa `pl2bat.bat` (habitualmente na directoria onde se encontra o `perl.exe`) que gera, a partir de `script.pl,` um ficheiro `script.bat` que pode ser invocado como

```
dos> script arg1 arg2 ...
```

O *Perl* dispõe de uma quantidade impressionante de páginas de manual ('manpages'). Em Unix acede-se a esses manuais com

```
unix> man perl
```

e em Windows com o utilitário `perldoc.bat` fazendo

```
dos> perldoc perl
```

A documentação encontra-se subdividida em sub-manuais cuja descrição sumária se encontra na página principal intitulada `perl`. Na distribuição para Windows os manuais estão disponíveis em formato `.html`.

O comando `perl` tem vários interruptores (ou opções): por exemplo

```
perl -w script.pl arg1 arg2 ...
```

executa `script.pl` e emite avisos ('warnings', daí `-w`) sobre, por exemplo, a utilização de valores de variáveis não inicializadas. Fazendo `perl -v` obtém-se a versão de *Perl* instalada. Pode-se fazer um pequeno script na linha de comando com o interruptor `-e`: por exemplo,

```
perl -e "print @ARGV;"a b c d 1245
```

imprime `abcd1245,` pois o vector `@ARGV` contém os argumentos fornecidos ao script (neste caso `a b c d 1245`).

**Recomenda-se vivamente a utilização do interruptor `-w`**, pelo menos na fase de

desenvolvimento dos programas.

## 2.2 Particularidades do *Perl*

Vejamos alguns pormenores básicos da programação em *Perl*.

- Basta um **editor de texto não formatado** 'freeware' (e.g. *Notepad, Wordpad, Editpad, NotetabLight*) para escrever um programa em *Perl*. Porém, pode utilizar-se editores potentes como o `emacs` e o `vi`, ou mesmo IDEs comerciais.

- Um **comentário** em *Perl* é o texto escrito entre `#` e o fim de uma linha. Existem formas mais complexas de comentar programas (veja 'plain old documentation', ou POD, na manpage `perlpod`).

- Uma instrução **termina** sempre com '`;`' excepto quando se tratar de um **bloco** de instruções, limitado por '{' e por '}', ou da última instrução de um bloco que não necessitam de terminar com '`;`' .

- Existem 'apenas' **três tipos** (importantes) de **variáveis** em *Perl*, completamente especificados pelo caracter com que começa a designação da variável:

  - as variáveis **escalares** começam **obrigatoriamente** por `$`. Estas variáveis podem conter números, strings ou referências (i.e., apontadores)
  - os **'arrays'** (vectores) começam **obrigatoriamente** por `@`. O **primeiro índice** de um array é zero (como no `C`). Como cada **elemento** de um array é **escalar**, para os referir inicia-se o nome por `$` (veja os exemplos mais abaixo)
  - os **'hashes'** (ou arrays associativos, ou dicionários) começam **obrigatoriamente** por `%`. Estas variáveis são colecções indexadas por strings, enquanto que os arrays são indexados por inteiros. Os hashes são de extrema utilidade prática na implementação de estruturas de dados complexas e de algoritmos.

  Os nomes de **subprogramas** (ou subrotinas, ou funções) são (quase sempre...) invocados com um `&` no início.

  Assim, **num mesmo programa** em *Perl* podem existir, sem qualquer conflito, os nomes `$nome`, `@nome`, `%nome` e `&nome`. **Representam entidades diferentes** entre si! Ainda pode existir a entidade `*nome`, denominada **'typeglob'**, que é uma referência genérica para qualquer das anteriores variáveis: os typeglobs eram fundamentais no `perl4` que não possuía referências, mas são pouco importantes no actual `perl5`.

- O **tipo caligráfico** das variáveis é importante: o *Perl* **diferencia maiúsculas** e **minúsculas**. Por exemplo, `$xpto` é diferente de `$Xpto`.

- Existem dezenas de variáveis pré-definidas: quase todas são uma combinação do caracter de definição do tipo da variável (`$`, `@` ou `%`) com um ou mais caracteres não alfabéticos (e.g. `$"`, `@_`, ...). A memorização destas variáveis é uma das grandes dores de cabeça para os iniciados no *Perl*. Muito importantes são `$_` e `@_`, pois são utilizadas, por defeito, por muitas funções quando estas são invocadas sem um argumento.

- **Não é obrigatória** a **declaração de variáveis** em *Perl* a não ser que, no início do programa, se use o pragma (ou 'directiva' para o interpretador) `use strict;`. Por exemplo, a instrução `$iarr[1000]=1999;` implicitamente declara e cria o array `$iarr`, e coloca o valor 1999 na posição 1001 (o início dos arrays é na posição 0).

- A escrita de subprogramas (feita segundo o padrão `sub subnome { instruções }`, onde `subnome` é a sua denominação) pode aparecer em qualquer parte do ficheiro de código; o interpretador faz duas passagens e assim permite a definição diferida de subrotinas.

- Existem vários tipos de strings: as escritas entre aspas `"...."` e as escritas entre plicas `'....'` são as mais importantes. Nas primeiras, é feita interpolação de variáveis escalares, de arrays e de caracteres 'escapados' (por exemplo `'\n'` indica o fim da linha e `'\t'` é um `tab`), enquanto que nas segundas não é feita qualquer interpolação. Por interpolação entende-se a substituição da variável embebida no texto da string. Veja-se o seguinte exemplo:

  ```
  $foo = 'Bobi';
  print "$foo fugiu!\n";    # imprime 'Bobi fugiu!'  e termina a linha
  print '$foo fugiu!\n';    # imprime '$foo fugiu!\n' e não muda de linha
  ```

  Existem formas alternativas de escrever strings: `'...'` e `"..."`são apenas duas delas.

  Igualando uma variável escalar a uma string-comando (string contendo um comando de sistema válido) escrita entre `` ` `` (acentos graves), o comando é executado e o resultado é atribuído à variável. É feita interpolação de variáveis nestas strings. Por exemplo;

  ```
  $command = 'dir';       # executa dir e guarda o resultado
  print `$command`;
  ```

  é equivalente a executar o comando `dir` (em DOS/Windows).

  Para executar um comando no sistema também se pode usar `system`. Por exemplo, se se quiser apagar o ficheiro `xpto.txt` na actual directoria de trabalho executa-se `system 'del xpto.txt';` em DOS (ou `system 'rm xpto.txt';` em Unix).

- O *Perl* é **liberal** na **sintaxe** de invocação de funções: por exemplo, para calcular o seno de `$num` (em radianos) pode usar-se `sin($num);` ou `sin $num;` (os parêntesis deverão ser utilizados sempre que houver dúvidas na precedência de operações).

- Uma das tarefas rotineiras onde o *Perl* brilha é na leitura, interpretação e transformação de ficheiros de texto. Estas tarefas são feitas normalmente com pouco código e usam exaustivamente **expressões regulares**.

  Suponha que existe, na directoria de trabalho, um ficheiro `texto.txt` que se pretende ler e processar. O seguinte código abre o ficheiro para leitura e coloca-o integralmente no array `@texto`. Cada elemento deste array irá conter uma linha do ficheiro

  ```
  open(FILEIN,"<texto.txt");    # O sinal < indica leitura do ficheiro
  @texto = <FILEIN>;            # Carrega o ficheiro no array @texto
  ```

  'Et voilá', está feito. O literal `FILEIN` é equivalente a um apontador para ficheiro em C, sendo denominado 'filehandle', ou simplesmente 'handle' em *Perl*.

  É igualmente simples escrever um array para dentro de um ficheiro. Isso será discutido mais adiante.

O seguinte bloco de código e o respectivo 'output' exemplificam alguns dos aspectos que discutimos. Pretende-se apenas dar um 'cheirinho' das facilidades do *Perl*.

```
#!/usr/bin/perl
# use a linha anterior para auto-executar o script em ambientes Unix/Linux

# Escalares

$vintpos    = 333;          # inteiro positivo...
$vintneg    = -10;          # e negativo
$vflsimple  = 12.3456;      # real em vírgula flutuante...
$vflscient  = -9.872e+10;   # e em notação científica
$vstr       = "string_1";   # string
$sref       = \$vintpos ;   # referência para $vintpos

# A numeração dos 'prints' é para facilitar a leitura

print "(1) $vstr interpolada\n";         # interpolação
print '(2) $vstr nao interpolada\n';     # sem interpolação
print "\n";
print;                                   # não imprime nada pois $_="" ...
$_ = '(3) Eu existo e sou $_';
print;                                   # ...mas agora $_ contém texto!
print "\n";                              # muda de linha

@varr = ('abc', "jk#mk", -12 , "jhj");   # inicializa o array @varr
print '(4) ' . $varr[1] . "\n";          # escreve o segundo elemento de @varr

%hash = ( 'meu', 'carro', 'teu',
    'piano' , 'dele' , 200, 1000, 'uff');  # inicializa um hash em 2 linhas
%hash = ( 'meu' => 'carro', 'teu' => 'piano',
    'dele' => 200, 1000 => 'uff');         # '=>' é equivalente a ',' em listas
# Esre código é equivalente a
# $hash{'meu'}='carro';
# $hash{'teu'}='piano';
# $hash{'dele'}=200;
# $hash{'1000'}='uff';

print '(5) ' . $hash{teu} . "\n";      # imprime o elemento da chave 'teu'
                                       # concatenação com '.'
$" = " ][ ";           # $" separa elementos na escrita de @arrays interpolados
print "(6) @varr\n";   # ... a variável $" por defeito era um espaço

@varr = %hash;         # @varr 'apanha' todo o hash
print "(7) @varr\n";

@varr = keys %hash;    # @varr 'apanha' as chaves do hash
print "(8) @varr\n";

@varr = values %hash;  # @varr 'apanha' os valores no hash
print "(9) @varr\n";

print '(10) ', quadrado(5);   # imprime resultado de função
                              # ainda não declarada
```

```
print "\n(11) ", $sref, " lixo\n";      # imprime referência em código 'interno'...
print "(12) ", $$sref, ' imprimiu $vintpos';  # 'de-refere', i.e., escreve $vintpos
print "(13) ", $vintneg + $vflsimple + $vflscient - $hash{'dele'}, "\n";
                                          # Cálculo numérico
print '(14)', $hash{'meu'} . $hash{1000} ;    # Concatenação de strings com '.'

# subprograma 'pós-declarado'
#
sub quadrado {
    my($x) = @_;     # nas subrotinas o array especial @_ ...
                     # ... serve para passar os argumentos
    return $x**2;    # 'return' devolve o valor
    $x**2;           # equivalente: sem um 'return' a subrotina devolve o ...
                     # ... último valor calculado no seu corpo.
}
```

O resultado da execução deste código com

```
dos> perl code.pl
```

(supondo que o script com o código é denominado `code.pl`) resulta em

```
(1) string_1 interpolada
(2) $vstr nao interpolada\n
(3) Eu existo e sou $_
(4) jk#mk
(5) piano
(6) abc ][ jk#mk ][ -12 ][ jhj
(7) dele ][ 200 ][ meu ][ carro ][ 1000 ][ uff ][ teu ][ piano
(8) dele ][ meu ][ 1000 ][ teu
(9) 200 ][ carro ][ uff ][ piano
(10) 25
(11) SCALAR(0x1aa530c) lixo
(12) 333 imprimiu $vintpos
(13) -98720000197.6544
(14) carro uff
```

Compare estes resultados com o código (a numeração nestas linhas está de acordo com as instruçoes `print` do código) e tente correlacioná-los.

## 2.3 'Manpages' do Perl

Eis (em duas tabelas) uma lista com as 'manpages' exixtentes na distribuição 'oficial' do *Perl* e uma breve descrição de cada uma. Apenas as mais importantes são referidas (faça `perldoc perl` para ver a lista completa). Sugere-se que consulte as manpages no formato `html` utilizando um 'browser'. Comece pelo ficheiro `PERLROOT/html/index.html` (onde `PERLROOT` é a raiz da instalação do *Perl* no seu sistema.)

Após esta introdução um pouco 'ad-hoc', passa-se a descrever o *Perl* de uma forma sistemática.

| | |
|---|---|
| perl | Introdução ao Perl |
| perldelta | Alterações no Perl relativamente à anterior versão |
| perlfaq | FAQ ('Frequently Asked Questions') sobre o Perl |
| perltoc | Índice da documentação do Perl |
| perldata | Estruturas de dados em Perl |
| perlsyn | Sintaxe do Perl |
| perlop | Operadores e regras de precedência em Perl |
| perlre | Expressões regulares em Perl |
| perlrun | Execução do Perl e opções ('command switches') |
| perlfunc | Funções nativas (intrínsecas) em Perl |
| perlopentut | Tutorial sobre a função open() em Perl |
| perlvar | Variáveis pré-definidas em Perl |
| perlsub | Subrotinas (ou procedimentos) em Perl |
| perlmod | Funcionamento de módulos em Perl |
| perlmodlib | Como escrever e usar módulos em Perl |
| perlreftut | Breve introdução às referências em Perl |
| perlref | Referências em Perl, o resto da história |
| perldsc | Introdução às estruturas de dados em Perl |
| perllol | Estruturas de dados em Perl: arrays de arrays |
| perlboot | Tutorial de Orientação para Objectos (OO) em Perl |
| perltoot | Tutorial de OO em Perl, parte 1 |
| perltootc | Tutorial de OO em Perl, parte 2 |
| perlobj | Objectos em Perl |
| perlbot | Truques e exemplos de OO em Perl |

Tabela 1: Páginas do manual do Perl.

| | |
|---|---|
| perlipc | Comunicação entre processos em Perl |
| perllexwarn | Avisos em Perl e seu controle |
| perldebug | 'Debugging' em Perl |
| perldiag | Mensagens de diagnóstico em Perl |
| perlnumber | Semântica dos números em Perl |
| perlsec | Segurança no Perl |
| perltrap | Armadilhas do Perl para tansos... |
| perlport | Guia de portabilidade do Perl |
| perlstyle | Guia de estilo da programação em Perl |
| perlpod | Documentação embebida em código Perl (POD) |
| perldebguts | Detalhes acerca do debugging de código |
| perldos | Notas sobre o Perl em DOS |
| perlwin32 | Notas sobre o Perl em Windows |

Tabela 2: Páginas do manual do Perl (cont.).

# 3 Variáveis

Os tipos de variáveis disponíveis, são um aspecto fundamental em qualquer linguagem de programação. O *Perl* suporta um conjunto de tipos de variáveis, de operadores e de funções bastante rico. Fazendo a composição dos tipos básicos (escalares, arrays e hashes) através do uso de referências, consegue-se implementar virtualmente qualquer estrutura de dados. Além disso, o *Perl* é inteligente (e liberal!) na interpretação das variáveis: por exemplo, considera como números strings que possam representar de facto um número (e.g. `'-3.25e-04'`). Contrariamente às linguagens fortemente restritivas na sintaxe (como o Pascal, o Java e, em menor grau, o C), onde é necessário fazer 'casts' entre tipos de variáveis diferentes, quase sempre o *Perl* (que não tem 'casts') consegue 'acertar' no tipo de variável subentendido pelo utilizador e adequado à tarefa que se pretende realizar.

As variáveis em *Perl* são criadas automaticamente na primeira utilização. Por exemplo `$x = 10.0;` cria a variável escalar `$x`, `$y[23] = "zzz";` cria a variável do tipo array `@y` com 24 elementos (de 0 a 23). Da mesma forma `$z{'primeiro'} = 1.000;` cria o hash `%z` com uma única chave `'primeiro'` e um único único elemento `1,000`. A gestão de memória é feita automaticamente.

Podem ser inicializadas várias variáveis simultaneamente com o mesmo valor

```
$a= $b= $c=0;
@aa = @bb = (1,2,'hj');
```

Para eliminar um elemento de um array ou de um hash utiliza-se a função `delete` (e.g., `delete $y[23];`). Para descobrir se uma dada variável existe, usa-se o operador `defined` (e.g., `defined($x)`).

## 3.1 Escalares

Um escalar em *Perl* tem um nome obrigatoriamente começado por `$`. Examine os seguintes exemplos comentados de escalares e de operações com escalares (são utilizados arrays e hashes nos exemplos, pois estes contêm elementos escalares):

```
# script denominado 'z.pl'

$integ = 100;                   # inteiro
$float = 1.33;                  # float
$str1  = "xpto";                # string
$str2  = "3e3";                 # string numérica, vale 3000
@vector = (1,2,3);              # array
%hmine = (1,2,"a","ss");        # hash
$sref  = \$integ;               # referência para $integ
$aref  = \@vector;              # referência para @vector
$href  = \%hmine;               # referência para %hmine
$anonrefa = ['a','b','c'];      # referência anónima para array
$anonrefb = {'a',1,'b',10};     # referência anónima para hash
print $$anonrefa[1], "\n";      # de-referem-se as referências
print ${$anonrefb}{'b'}, "\n";  # ... e imprime-se b e 10
print $integ + $float, "\n";    # imprime a soma 101.33
print $integ . $float, "\n";    # imprime a concatenação 1001.33
                                # ... considerando os números como
```

```
                                # ... strings
print $integ + $str1, "\n";     # imprime 100 + 0 pois $str1 não
                                # ... é reconhecida como número
print $integ + $str2, "\n";     # imprime 3100 pois $str2="3e3"
                                # ... é reconhecida como número
$reftoref = \$sref;             # referência para referência
print $$$reftoref,"\n";         # dupla de-referência igual a $integ
delete $hmine{1};               # elimina valores de hash e array
delete $$anonrefa[1];
```

O resultado da execução deste código é

```
b
10
101.33
1001.33
100
3100
```

Nestes exemplos mostra-se que uma variável escalar pode conter valores numéricos, strings e referências. Além disso, pode ser confirmado que o *Perl* é liberal: se se utilizar uma string onde deveria estar um número, o interpretador verifica se a string representa um número (o que acontece com `$str2=3000="3e3"`) e, no caso afirmativo, considera-a como um número com esse valor. Se a string não tem uma representação numérica válida, é-lhe atribuído o valor zero 0 e efectuado, na mesma, o cálculo!

Se quiser ser avisado destas 'liberdades' do *Perl* (uma string considerada como 0) invoque-o como `perl -w z.pl`: o interpretador, quando chega à linha

```
print $integ + $str1, "\n";
```

produz o seguinte aviso:

```
Argument "xpto" isn't numeric in addition (+) at z.pl line 18.
```

Um array `@vector` e um hash `%hmine` são preenchidos colocando uma lista de elementos, separados por vírgulas ou por `'=>'`, entre parêntesis `(...,...=>,...)`.

Como o hash contém pares chave-valor, o número de elementos no preenchimento de `%hmine` deverá ser par: mas se fôr ímpar, o *Perl* não se lamenta e iguala o conteúdo da última chave a `""` (string nula). Se fôr invocado com o interruptor `-w` dá o seguinte aviso:

```
Odd number of elements in hash assignment at z.pl line 7.
```

### 3.1.1 Notas sobre referências

No anterior bloco de código, `$sref`, `$aref` e `$href` são três referências (ou apontadores), respectivamente para uma variável escalar, um array e um hash. São construídas pondo `'\'` em frente das variáveis que para onde vão apontar.

Por sua vez, `$anonrefa` e `$anonrefb` são duas referências anónimas para um array e para um hash. Neste caso, as referências são criadas igualando-as a uma lista de valores entre parêntesis rectos `[...]` ou a uma lista entre chavetas `{...}`, consoante o se pretenda referir um array ou um hash.

A de-referência faz-se antecedendo o apontador com o caracter 'especial' que identifica o tipo da variável apontada. Assim, atendendo ao anterior bloco de código, verifica-se que:

`$$sref` equivale a `${$sref}` é igual a `$integ`

e que:

`$$anonref{'b'}` equivale a `${$anonref}{'b'}` e equivale a `$hmine{'b'}`

A utilização ou não de chavetas para de-referir fica ao gosto do programador, embora a sua utilização clarifique o código. Quando se utilizam chavetas, diz-se que se utilizou um `BLOCO` como o nome da variável (note que um bloco em *Perl* designa o código entre {...}.

Podem ser feitas de-referências sucessivas. Se fizermos

`$reftoref = \$sref;`

então

`$$$reftoref` equivale a `$integ;`

e tem o valor 100.

Existe outra forma de de-referir arrays ou hashes: o operador seta `'->'`. Lembre-se que `$aref` e `$href` eram referências respectivamente para um array e para um hash. Eis uma repetição do código:

```
@vector = (1,2,3);              # array
%hmine = (1,2,"a","ss");        # hash
$aref  = \@vector;              # referência para @vector
$href  = \%hmine;               # referência para %hmine
```

Então, para obter `$vector[2]` (que vale 3) com a referência `$aref` pode utilizar-se `$$aref[2]`, `${$aref}[2]` ou `$aref->[2]`. Da mesma forma `$$href{"a"}`, `${$href}{"a"}` e `$href->{"a"}` devolvem a string `"ss"` no hash `%hmine`. O operador `'->'` também é utilizado em arrays multidimensionais (ou matrizes) e em outras estruturas de dados complexas que discutiremos mais adiante.

As referências que acabámos de descrever, criadas com `'\'`, são denominadas **referências fortes** ('hard'). Há outro tipo de referências: as **referências simbólicas.**

Quando se tenta de-referir um valor que não é uma referência forte, esse valor é tratado como uma referência simbólica com o **nome** da variável. Nas versões anteriores ao *Perl*5 este era o único tipo de referências. Veja os seguintes exemplos:

```
$nome = "x";                # atribui a string "x" a $nome
$$nome = 1;                 # $x = 1;
${$nome} = 2;               # $x = 2;
${$nome x 3} = 3;           # $xxx = 3;
$nome->[0] = 4;             # cria array @x e atribui 4 a $x[0]
@$nome = ();                # limpa @x
&$nome();                   # invoca &x()
$nome->{"tu"} = "Maria" ;   # cria hash %x e atribui "Maria" a $x{"tu"}
```

As referências simbólicas são extremamente poderosas, mas podem originar 'bugs' difíceis de detectar. Pode impedir-se o seu uso com o pragma `use strict 'refs';` (só permite o uso de referências fortes) e voltar a permitir referências simbólicas (por exemplo dentro de um bloco) com o pragma `no strict 'refs';`.

Refira-se que variáveis locais declaradas com `my` (veja, mais adiante, a discussão de subprogramas e variáveis locais) não são visíveis na tabela de símbolos: assim, não podem ser acedidas por referências simbólicas.

### 3.1.2 'Strings', listas e expressões regulares. Interpolação

Existem diferentes tipos de string. Apenas apresentámos a sua sintaxe habitual (e.g., `'...'`, `"..."`): porém, existem formas alternativas de criá-las que são, por vezes, mais legíveis. O mesmo pode ser dito relativamente a listas e a expressões regulares.

Na tabela 3 são enumeradas as alternativas existentes para criação/especificação de strings, de expressões regulares e de listas.

| **Habitual** | **Genérica** | **Significado** | **Interpolação** |
|---|---|---|---|
| `''` | `q//` | Literal | Não |
| `""` | `qq//` | Literal | Sim |
| `` ` ` `` | `qx//` | Comando | Sim |
| `()` | `qw//` | Lista de palavras | Não |
| `//` | `m//` | Expressão Regular | Sim |
| `s///` | `s///` | Substituição | Sim |
| `y///` | `tr///` | Transliteração | Não |

Tabela 3: Formas equivalentes de especificação de strings e listas.

Nas formas alternativas, o par ou trio de '/' pode ser substituído por qualquer par ou trio de caracteres não alfanuméricos (i.e., não incluídos em `a...zA...Z0...9` e `_`) e que não sejam 'espaços' (e.g., `' '`, `\n` ou `\t`). Se forem utilizados parêntesis na sintaxe `'/.../'`, o caracter de abertura será escolhido de entre `([{` e o caracter de fecho será o parêntesis complementar no grupo `)]}`.

Pode utilizar-se parêntesis em `s///` ou em `tr///` desde que se usem **dois pares compatíveis** de parêntesis, como por exemplo em `s{abc}{xyz}` ou `tr [a-z] /A-Z/`. Repare que neste último exemplo os dois pares de caracteres são diferentes (o que é permitido) e que se pode incluir espaços entre as três partes que compõem o operador.

Para clarificar um pouco estas equivalências, mostram-se alguns exemplos ('emparelhados' consecutivamente e partilhando a mesma variável).

```
$xxx  = 'malandro';                # palavra para interpolar
$stni = 'Bom dia!';                # string não interpolada
$stni = q(Bom dia!);               # parêntesis a abrir '(' e a fechar ')'
$sti  = "O Manuel é $xxx\n" ;      # string interpolada
$sti  = qq   |O Manuel é $xxx\n|;  # o caracter '|' abre e fecha.
                                   # qq e '|' estão separados por ' '
print $stni, qq/\n/ , $sti , qq[\n];
                                 # qualquer forma é admitida em argumentos
$com  = `cat $0`;                # $com contém a listagem do actual script...
$com  = qx?cat $0?               # ... cujo nome está contido em $0
$com  = `type $0`;               # o mesmo em Windows!
$com  = qx?type $0?
@lista = ("a",bb,"ccc","dddd");  # perl -w dá aviso sobre o termo bb sem aspas
@lista = qw [a bb ccc dddd];     # não é preciso usar '' ou aspas "" nos elementos
$troca = y#a-z#A-Z#;             # 'maiusculiza' a variável $troca...
$troca = tr [a-z]
            [A-Z];               # o 'newline' (um espaço...) pode ocorrer entre as
                                 # ...partes dos operadores tradução ou substituição
```

Pode interpolar-se variáveis ou arrays em strings `"..."`, expressões regulares `//`, etc... (veja a tabela 3). Os elementos dos arrays são concatenados entre si com os caracteres contidos na variável interna `$"` que normalmente (e por defeito) é um espaço `' '`. Assim,

```
$temp=join($",@myarr);
print $temp;
```

equivale a `print @myarr;`.

O *Perl* disponibiliza operadores para transliterar letras de maiúscula para minúscula e vice-versa:

- **lc** `STRING`
  devolve a string `STRING` em minúsculas (ou `$_` se `STRING` fôr omitida).
- **lcfirst** `STRING`
  devolve `STRING` (ou `$_` se `STRING` fôr omitida) com o *primeiro caracter* como minúscula.
- **quotemeta** `EXPR`
  devolve `EXPR` (ou `$_` se `EXPR` fôr omitida) com todos os caracteres não alfanuméricos escapados com `'\'`. Veja `\Q` e `\E` na tabela de caracteres escapados abaixo.
- **uc** `STRING`
  devolve `STRING` em maiúsculas (ou `$_` se `STRING` fôr omitida).
- **ucfirst** `STRING`
  devolve `STRING` (ou `$_` se `STRING` fôr omitida) com os *primeiros caracteres* em maiúsculas.

Estas transformações podem ser efectuadas em *strings interpoladas* com caracteres escapados. Na seguinte tabela, apresentam-se estes caracteres bem como outros caracteres escapados não relacionados com maiúsculas/minúsculas:

| **Escape** | **Significado** |
|---|---|
| `\cC` | control-C |
| `\\` | 'backslash' ou `'\'` |
| `\"` | aspa |
| `\u` | força o próximo caracter a maiúscula |
| `\l` | força o próximo caracter a minúscula |
| `\U` | força todos os caracteres seguintes a maiúscula (termina com `\E`) |
| `\L` | força todos os caracteres seguintes a minúscula (termina com `\E`) |
| `\Q` | 'escapa' os caracteres não alfanuméricos seguintes (termina com `\E`) |
| `\E` | termina `\U`,`\L` ou `\Q` |

O *Perl* possui mais algumas funções de grande importância na manipulação de strings.

- **chr** `NUMBER`
  Esta função devolve o caracter ASCII representado pelo número. Por exemplo, `chr(66)` dá `"B"`. A função inversa é `ord()`.
- **ord** `EXPR`
  Esta função devolve o valor numérico do primeiro caracter ASCII em `EXPR`. Se `EXPR` fôr omitida, usa `$_` por defeito. Por exemplo, `ord("D")` é 68. A função inversa é `chr()`.

- **substr** `EXPR`, `OFFSET`, `LENGTH`
  **substr** `EXPR`, `OFFSET`
  Esta função extrai uma sub-string da string dada pela expressão `EXPR` e devolve-a. A sub-string começa no caracter especificado pelo número `OFFSET`, contado a partir do início da string (supondo que `$[=0`). Se `OFFSET` fôr negativo, começa-se a contar do fim da string. `LENGTH` dá o tamanho da string a extrair: se não fôr especificado, a extracção é feita até ao fim da string. Se `LENGTH` fôr negativo, a dimensão da string a extrair é calculada de forma a terminar `|LENGTH|` caracteres antes do fim da string.

  Se `substr()` aparecer no membro esquerdo de uma atribuição, `EXPR` deverá ser uma variável válida. Por exemplo
  ```
  substr($nome,0,0) = "Manuel ";
  ```
  junta `"Manuel "` ao início de `$nome`.

- **index** `STR`, `SUBSTR`, `POSITION`
  **index** `STR`, `SUBSTR`
  Esta função devolve a posição da **primeira** ocorrência da sub-string `SUBSTR` na string `STR`. Se a posição `POSITION` fôr especificada, é nela que começa a pesquisa. Se `SUBSTR` não fôr encontrada, a função devolve a posição base especificada pelo valor de `$[` (habitualmente 0) menos 1 (assim, habitualmente é devolvido -1).

- **rindex** `STR`, `SUBSTR`, `POSITION`
  **index** `STR`, `SUBSTR`
  Esta função funciona como `index`,só que devolve a posição da **última** ocorrência da sub-string `SUBSTR` na string `STR`. Se `POSITION` fôr especificada, é a última posição na `STR` a ser pesquisada. Se `SUBSTR` não fôr encontrada, a função devolve o valor `$[-1`.

### 3.1.3 Operadores aplicáveis a escalares

O *Perl* tem um leque de operadores extenso e completo. Vejamos alguns operadores aplicáveis a escalares.

- Operadores algébricos `+, -, /, *, **, %` onde `**` é a exponenciação e `%` é a divisão modular. Aplicam-se a números. As strings são 'convertidas' para escalares numéricos conforme já foi explicado. Vejamos alguns exemplos.
  ```
  $a = 10.0;
  $b = 5;
  $c = $a * $b;      # $c vale 50.0
  ```

- Os operadores de concatenação `'.'` e de repetição (ou replicação) `'x'` estão disponíveis para strings. Assim:
  ```
  $a = 10.0;
  $b = 5;
  $c = $a .  $b;      # $c vale 10.05
  $d = '-';
  print $d x 80;    # imprime uma linha com oitenta '-'
  ```
  O operador de replicação tem uma aplicação interessante em arrays. O código `@z = (0) x 11;` cria o array `@z` preenchido com 11 zeros (posições de 0 a 10).

- Os incrementadores `++` e `--` comportam-se tal como no C.

```
$a = 10;
++$a;           # $a vale 11(equivale a $a++)
$a--;           # $a vale de novo 10 (equivale a --$a)
$b = $a++;      # $a vale 11 e $b vale 10 (pós-incremento)
$c = --$a;      # $a e $c valem 10 (pré-decremento)
$s = "cab";
$s++;           # $s vale "cac"...  O quê?
```

  Os incrementadores são mágicos! Veja o seguinte código:

```
$a="haa";
++$a;
print $a;
```

  Incrementar uma string? Pois é, se executar este código, vai ver que no final `$a="hab"` tendo o *Perl* tratado a string com caracteres do alfabeto como se fosse um número escrito numa base 26! Experimente executar o anterior código com `$a="ZZZ"`...

- Existem os operadores `$var op= $value` que equivalem a `$var = $var op $value`, onde `op` é um dos operadores `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `.=`, `x=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `&&=`, `||=`. Funcionam como no C.

```
$a=20;
$b=4;
$b += $a      # $b vale 24
$b .= $a      # $b vale agora 244
$b /= $a      # $b vale 61
```

- Estão disponíveis os habituais operadores de comparação `<, >, >=, <=, ==, !=,` e `<=>`. Fazem comparação numérica. Existem operadores equivalentes (pela mesma ordem) para comparação de strings que são `lt, gt, ge, le, eq, ne,` e `cmp`. Oops! O C não tem o operador de comparação `'<=>'` (que os 'perlistas' chamam carinhosamente de 'starship operator' ...). Quando dois escalares são comparados (por exemplo, `$res = $a <=> $b`) o resultado `$res` é -1 se `$a<$b`, 0 se `$a==$b` e +1 se `$a>$b`. O operador `cmp` tem o mesmo comportamento mas compara strings.

- Existem os operadores lógicos `&&`, `||` e `!` (respectivamente 'e', 'ou', e 'não'). Existem ainda os operadores `and`, `or` e `not` com o mesmo significado mas com precedência mais fraca do que os anteriores.

- Como em C, o operador condicional ternário `$value = cond ? true_value : false_value ;` permite atribuir a `$value` um dos valores `true_value` ou `false_value` consoante a condição `cond` seja verdadeira ou falsa. Por exemplo

```
$x = 10;
$z = $x>0 ?  "pos":  "not-pos";
```

  torna `$z` igual a `"pos"` pois `$x` é positivo.

- Há os operadores aplicáveis ao nível de bit `<<, >>, ~, &, |`, e `^` (respectivamente deslocamento à esquerda, deslocamento à direita, complemento, e, ou e ou exclusivo). A aplicação dos operadores de deslocamento é exemplificada de seguida:

```
$num = 16 ;
```

```
$dobro = $num << 1 ;        # shift de 1 posição à esquerda duplica
$metade = $num >> 1 ;       # shift de uma posição à direita divide por 2
```

- Os operadores `,` e `=>` são utilizados de forma equivalente na definição de listas (arrays ou hashes). O operador `=>` é particularmente útil na escrita de hashes. Veja mais abaixo.
- Há os operadores de coincidência `=~` e não coincidência `!~` aplicados em expressões regulares.
- A função `length EXPR` devolve o número de bytes de `EXPR` em contexto escalar. Se `EXPR` fôr omitida, aplica-se implicitamente a `$_`. Se `EXPR` fôr uma expressão complicada, deverá ser escrita entre parêntesis, i.e., usa-se `length(EXPR)`. Pode ser aplicada a strings para saber a sua dimensão.

Para ver uma listagem completa e a precedência dos operadores, faça `perldoc perlop` em Windows ou `man perlop` em Unix/Linux. A título de ajuda, leve em conta que os operadores 'copiados' do C mantêm a mesma precedência. Pode (e deve!) utilizar parêntesis sempre que tiver dúvidas!

### 3.1.4 Funções matemáticas

O *Perl* dispõe das funções matemáticas básicas. Para utilizar funções especializadas pode recorrer-se a módulos (ou bibliotecas) no CPAN. Eis uma lista das funções nativas.

- **abs** `VALOR`
  calcula o valor absoluto do argumento (ou de `$_ se` o argumento `VALOR` fôr omitido).
- **atan2** `Y, X`
  calcula o arco-tangente de `Y/X` no domínio $[-\pi, \pi]$. Para obter o valor de $\pi$ faz-se:
  `$pi = atan2(1,1) * 4;`
- **cos** `EXPR`
  calcula o coseno da expressão `EXPR` (expressa em radianos). Se `EXPR` fôr omitida, calcula `cos $_`.
- **hex** `EXPR`
  interpreta `EXPR` como uma string de caracteres hexadecimais e devolve o seu valor numérico (e.g., `hex "a0"` vale 160). Note que uma string hexadecimal pode começar por '0x' (isto é, `hex "0xa0"` vale também 160).
- **int** `VALOR`
  devolve a parte inteira do argumento (ou de `$_` se o argumento `VALOR` fôr omitido).
- **log** `EXPR`
  esta função calcula o logaritmo neperiano (de base $e$) de `EXPR`, ou de `$_` caso aquele argumento seja omitido.
- **oct** `EXPR`
  interpreta `EXPR` como uma string de caracteres octais (na base 8) e devolve o seu

valor numérico decimal (e.g., `oct "055"` e `oct '055'` valem 45). Uma string octal começa por '0'. Se a string começar por `"0x"` a função `oct()` reconhece-a como sendo um número em hexadecimal e converte correctamente o seu valor (e.g., `oct "0x55"` e `oct '0x55'` valem 85.)

- **rand** `EXPR`
  **rand**
  calcula um número fraccionário aleatório que segue uma distribuição uniforme entre 0 e `EXPR`. Se `EXPR` fôr omitida, devolve um valôr entre 0 e 1 (1 não está incluído).

- **sin** `EXPR`
  calcula o seno da expressão `EXPR` (expressa em radianos). Se `EXPR` fôr omitida, calcula `sin $_`.

- **sqrt** `EXPR`
  **sqrt**
  calcula a raiz quadrada de `EXPR`. Se `EXPR` fôr omitida calcula `sqrt $_`. Para calcular raízes cúbicas, etc, pode ser utilizado o operador exponenciação `**` com expoente fraccionário.

- **srand** `EXPR`
  **srand**
  inicializa o gerador de números aleatórios utilizado pela função `rand`. Se `EXPR` fôr omitida faz `srand(time)` (que é uma operação cujo resultado é bastante previsível: não a use se pretender cifrar algo importante).

- **time**
  imprime o número de segundos desde 1 de Janeiro de 1970 (actualmente quase $10^9$).

## 3.2 'Filehandles' (referências para ficheiros)

Os 'filehandles' (ou 'handles', para simplificar) são referências para ficheiros externos manipuladas nos scripts. São um tipo especial de variável, que não utiliza nenhum caracter especial como prefixo para o nome. É **recomendado** que o **handle** seja escrito em **maiúsculas.** Esta regra não é obrigatória, mas é seguida por quase toda a gente. A razão desta recomendação é que qualquer palavra em minúsculas potencialmente poderá, no futuro, vir a ser considerada um comando ou operador em *Perl*.

Cada ficheiro manipulado em *Perl* tem associado um filehandle (ou nome 'lógico') e um nome físico, relacionado com a posição do ficheiro na directoria do sistema. Essa associação é estabelecida quando o ficheiro é aberto. Os ficheiros podem ser abertos para leitura, para escrita, e para actualização ('appending'). Para ilustrar a utilização dos handles recorre-se a um pequeno bloco de código.

```
$fin = 'aqui.txt';
open(FILEIN,"$fin");    # abre o ficheiro 'aqui.txt' na actual directoria
                      # ...para leitura (modo de defeito). $fin é interpolado
open(FILEIN,"<$fin") || die("$fin não existe!\n") ;    # O mesmo...
open(FILEOUT,">d:/text/myout.txt");   # abre '...\myout.txt' para escrita
open(FILEOUT,">>d:/text/myout.txt");  # abre '...\myout.txt' para actualização
close(FILEIN);
close(FILEOUT);                       # fecha os ficheiros
```

É importante notar alguns aspectos. Primeiro, é feita interpolação de variáveis na string com o nome físico do ficheiro se ela fôr escrita entre `"..."`. Segundo, a indicação do modo de utilização do ficheiro é dada com $'<'$ para ler, e este é o modo de defeito, com $'>'$ para escrever)e com $'>>'$ para acrescentar. Terceiro, o caminho ('path') do ficheiro é escrito 'a la Unix', isto é, faz uso de `'/'` e não de `'\'` como em DOS/Windows. Note que isto é válido mesmo para este sistema operativo. Finalmente, embora se tenha fechado explicitamente os ficheiros com `close(FILEHANDLE)` no exemplo, não há problema em não o fazer pois, quando termina a execução do script, todos os ficheiros são fechados automaticamente.

Ainda relativamente ao anterior exemplo, há um outro aspecto importante a reter. A execução da função `die("...")` ('morre'...) termina um script e escreve para a consola a string dada no argumento. Acontece que na avaliação de expressões lógicas com várias cláusulas o *Perl* começa a partir da esquerda e termina a avaliação assim que conclui da verdade ou da falsidade da expressão. Ora `cond1 || cond2` é sempre verdade se `cond1` o fôr. Isto significa que o *Perl* só executa `die("...")` se falhar a abertura do ficheiro com `open(FILEIN,"<$fin")`, o que indica que ele não existe ou que o utilizador não tem privilégio para o abrir, em Unix ou noutro SO que respeite os administradores do sistema e a privacidade dos utilizadores `8-)`.

Este comportamento do *Perl*, denominado 'shortcut evaluation', é muito utilizado na prática.

E agora, como é que se pode ler de e escrever para ficheiros?

Para escrever utiliza-se a instrução

```
print FILEHANDLE 'string1', $var, ...  , "string2";
```

que, por defeito, quando `FILEHANDLE` é omitido, escreve para a consola, ou seja, para o filehandle `STDOUT` que se encontra automaticamente disponível durante a execução de programas. Os campos separados por `','` podem ser variáveis escalares ou strings, interpoláveis ou não. A interpolação de arrays é suportada em strings interpoláveis (e.g., `string2` na anterior instrução).

O *Perl* suporta as instruções de escrita formatada `printf` e `sprintf`.

`printf FILEHANDLE FORMATO, LISTA` escreve a string `FORMATO` com as variáveis de `LISTA` interpoladas de acordo com códigos semelhantes aos da instrução homónima em C.

`sprintf FORMATO, LISTA` devolve apenas a string. Por exemplo:

```
$nome_de_pessoa = 'Manel' ;
$sform = sprintf "Bom dia %s!\n", $nome_de_pessoa ;
```

escreve `"Bom dia Manel!`. O código `%s` indica que `$nome_de_pessoa` é uma string.

Consulte o manual para saber, em detalhe, quais os códigos de formatação disponíveis para as funções `printf` e `sprintf`.

Na leitura de dados de ficheiros manifesta-se a versatilidade do *Perl*. Para ler uma linha do ficheiro associado a `FILEIN` para a variável escalar `$line` faz-se

```
$line = <FILEIN>;
```

Para ler a totalidade do ficheiro associado a `FILEIN` para um array `@arrayfile` faz-se

```
@arrayfile = <FILEIN>;
```

Este é um exemplo da execução de uma mesma expressão 'à direita' (neste caso `<FILEIN>;`) em *contexto escalar ou em contexto de lista*. Consoante o tipo de variável que se encontra no membro esquerdo da atribuição (escalar ou array/lista), assim é o modo

como os dados são lidos de `FILEIN`.

Mais adiante, quando falarmos da execução de tarefas repetitivas em *Perl* (com `while`, `until`, `for`, `foreach`, ...) retomaremos o tópico da leitura de dados de ficheiros externos. Por exemplo, será ilustrada a leitura de ficheiros, caracter a caracter, com as funções `getc(FH)` e `eof(FH)` que, de uma forma semelhante ao C, respectivamente lêem um caracter do ficheiro associado ao filehandle `FH` ou detectam o seu fim ('end-of-file').

### 3.2.1 Operadores de teste a ficheiros

O *Perl* dispõe de operadores de teste aplicáveis a ficheiros. Permitem avaliar o tipo do ficheiro, as permissões de acesso, etc. São operadores unários. Eis alguns exemplos:

```
$dim = -s "c.pl";    # $dim guarda o tamanho do ficheiro c.pl na actual directoria
print "Ficheiro de texto\n" if -T "texto.txt";        # verifica se é ficheiro de texto
print "Ficheiro não existe\n" if !(-e "file.dat");    # verifica se file.dat não existe
```

A tabela seguinte enumera alguns destes operadores que testam, quer o ficheiro, quer as permissões do utilizador (são agrupados mais do que um em cada linha).

| Operadores | Significado |
|---|---|
| `-r`, `-w`, `-x`; `-o` | permissão de ler, escrever, executar; posse |
| `-R`, `-W`, `-X`; `-O` | permissão de ler, escrever, executar; posse |
| `-e`, `-z`; `-s` | existe, tem dimensão zero; devolve a dimensão |
| `-T`, `-B` | ficheiro de texto, ficheiro binário |
| `-M`, `-A` | tempo desde o/a último/a modificação, ou acesso |
| `-f`, `-d`, `-l` | é ficheiro, é directoria, é 'link' simbólico |
| `-b`, `-c` | ficheiro especial de blocos, ou de caracteres |

Estes operadores aplicam-se quer a nomes 'físicos' de ficheiros quer a filehandles.

### 3.2.2 Tópicos mais avançados na utilização de `<...>`

Viu-se a utilização do operador `<FHAND>` contendo um filehandle `FHAND` genérico, normalmente escrito em maiúsculas. Porém, esta não é a única forma de utilizar o 'interior' de `<...>`.

Se a string encerrada por este operador fôr uma variável escalar (por exemplo `<$foo>`) e contiver o nome do filehandle que se pretende aceder ou uma referência para ele, o *Perl* 'entende' e acede ao ficheiro associado. Veja o exemplo.

```
open(MYSELF,$0);      # abre o próprio script. $0 contém o nome do script
$fh = MYSELF;         # guarda o nome do filehandle...
$fh = \*MYSELF;       # ...ou a referência para o filehandle (é equivalente)
$line = <$fh>;        # lê a primeira linha do script
```

Se entre `<...>` estiver uma expressão que não seja um handle ou uma variável (mesmo que sejam só espaços! Por exemplo, no anterior código `$line = < $fh>;` já não era interpretado correctamente por causa do espaço extra), o *Perl* interpreta o interior de `<...>` como um 'padrão' de nomes de ficheiros na actual directoria. Por exemplo, executando `@perlfiles = <*.pl>;` coloca-se no array `@perlfiles` o nome de todos os scripts com sufixo `.pl` existentes na actual directoria.

### 3.2.3 Outras funções de acesso a ficheiros

O *Perl* possui mais algumas funções de leitura/escrita/processamento de ficheiros.

`seek FILEHANDLE, OFFSET, OFFSETMODE` coloca o ponto de acesso ao ficheiro referido por `FILEHANDLE` na posição especificada por `OFFSET` (a posição inicial no ficheiro é zero). O modo `OFFSETMODE` igual a 0/1/2 indica, respectivamente, se `OFFSET` é contado a partir do início, da actual posição, ou do fim do ficheiro. A unidade de contagem são *bytes*.

`tell FILEHANDLE` devolve a actual posição de leitura no `FILEHANDLE` em bytes, a contar de 0. Se `FILEHANDLE` fôr omitido aplica-se ao último ficheiro que foi lido.

`read FILEHANDLE, ESCALAR, TAMANHO, OFFSET` tenta ler `TAMANHO` bytes de dados do ficheiro associado a `FILEHANDLE` e armazená-los na variável `ESCALAR`. A função devolve o número de bytes efectivamente lidos ou 0, caso encontre o fim do ficheiro. Devolve `undef` se ocorrer um erro. O valor de `OFFSET` (se utilizado) especifica a posição na string `ESCALAR` onde se começa a escrever os dados.

`write FILEHANDLE` escreve um registo formatado de acordo com a definição de 'formats'. Veja o manual `perlform`.

## 3.3 Dirhandles (referências para directorias)

O *Perl* possui um conjunto de funções para lidar com directorias que se descreve sucintamente.

`rmdir DIRNAME` apaga a directoria especificada por `DIRNAME` se ela estiver vazia. Se `rmdir` fôr bem sucedida, devolve 1; se não, devolve 0 e escreve um código de erro em `$!`. Se `DIRNAME` fôr omitido, `rmdir` opera sobre o conteúdo de `$_`.

`chdir EXPR` muda a actual directoria de trabalho para `EXPR`, se possível. Se `EXPR` fôr omitida, muda para a `'home'` do utilizador. `chdir` devolve 1 em caso de sucesso. Em caso de insucesso devolve 0 e escreve um código de erro em `$!`.

`mkdir DIRNAME, MODO` cria a directoria especificada por `DIRNAME` com o modo especificado por `MODO`. Devolve 1/0 em caso de sucesso/insucesso. No insucesso, escreve o código de erro em `$!`.

`opendir DIRHANDLE, EXPR` abre a directoria de nome `EXPR` e associa-a à referência lógica `DIRHANDLE`, para posterior processamento por `readdir`, `telldir`, `seekdir`, `rewinddir` e `closedir`. A função devolve verdade, em contexto lógico, se foi bem sucedida. As 'dirhandles' têm uma tabela de nomes independente da tabela de nomes dos filehandles.

`closedir DIRHANDLE` fecha a directoria aberta com `opendir`.

`telldir` e `seekdir` estabelecem o posicionamento de leitura da directoria por parte de `readdir`. `rewinddir` coloca a actual posição de leitura da função `readdir` no início, i.e., em `DIRHANDLE`.

A função mais importante é `readdir DIRHANDLE`. Por exemplo, para processar a actual directoria de trabalho pode fazer-se:

```
opendir TRABALHO, "." or die "A directoria '.' não existe ???: $!";
@todos = readdir TRABALHO;              # lê a directoria para um array
@textos = grep -T, readdir TRABALHO;    # lê só ficheiros de texto
```

```
closedir TRABALHO;                    # fecha a dirhandle
print "@todos\n";                     # lista-a (equivale a 'dir' ou 'ls')
```

### 3.4 'Arrays' (ou vectores ou listas)

Os arrays são variáveis cujo nome começa por `@` e consistem de uma lista de elementos indexados por um número inteiro. Podem ser criados explicitamente com uma lista de elementos separados por vírgulas como, por exemplo, `@vector  = (1,2,3);`

A primeira posição de um array é 0: este índice pode ser mudado com a variável especial `$[` (e.g., `$[=1;`) mas isto não é recomendado. Não é necessário declarar arrays, a não ser que seja usado o pragma `use strict;` que força a declaração de todas as variáveis. Pode eliminar-se elementos de arrays ou de hashes com a função `delete` (e.g `delete $myarray[3];` ou `delete $myhash{'boo'};`). O `delete` de um elemento de um array não lhe altera a dimensão, pois o elemento apagado fica 'undefined mas a 'célula' não desaparece. Se quiser eliminar uma célula de um array use `splice`.

Ao igualar um escalar a um array, atribui-se-lhe o número de elementos (ou dimensão) do array. Por exemplo

```
$tamanho = @vector;
```

faz com que `$tamanho = 3` (pois `@vector` tem 3 elementos). Cada array tem associada uma variável escalar especial que contém o maior índice existente no array. Para um array denominado `@arrxpto` essa variável é `$#arrxpto` (substitui-se `@` por `$#` na denominação do array). Assim, `$#arrxpto` vale dois no exemplo (o array `@arrxpto` tem elementos nas posições 0, 1 e 2).

É fácil copiar arrays. Suponha que temos um array

```
@bb = ("00", 1 , "22", 333, 4);
```

e que se faz a seguinte atribuição

```
@aa = @bb;
```

A atribuição entre dois arrays `@aa = @bb;` iguala o array `@aa` a `@bb`. Nesta igualdade, a **duplicação é feita por valor** e não por referência (contrariamente, por exemplo, ao `python`, onde quando dois arrays são igualados a operação é feita por referência, a não ser que se force uma duplicação). Isso pode ser verificado com o seguinte código

```
@arr = (0,1,2,3,4);
@barr = @arr;
$arr[2] = 'foo';
print "@barr\n";      # imprime 0 1 2 3 4
print "@arr \n";      # imprime 0 1 foo 3 4
```

Como se vê, a alteração em `$arr[2]` não alterou o valor de `$barr[2]` que tinha sido previamente copiado.

Um array é uma lista. A atribuição `@aa = @bb` é apenas a atribuição implícita de uma lista a outra. A cópia pode ser realizada entre listas de escalares. Por exemplo

```
($a,$b,$c) = ($x,$y,$z);
```

realiza três atribuições de escalares simultaneamente. Este mecanismo é muito útil para trocar os valores de dois escalares (o denominado 'swap') fazendo-se

```
($a,$b) = ($b,$a);      # Troca os valores de $a e $b duma penada!
```

Podem retirar-se fatias (*'slices'*) de arrays com o operador `'..'` denominado 'operador

gama' (de 'range' em Inglês, não da letra grega...) Veja o exemplo:

```
@arr=(0,1,2,3,4,5,6);
@fatia = @arr[1..3];     # @fatia = (1,2,3);
push @arr, @fatia;     # copia @fatia para o fim de @arr
                     # equivalente a push @arr, (1,2,3);
```

No final, `@arr=(0,1,2,3,4,5,6,1,2,3)`, isto é, tem 10 elementos. A função `push` concatena duas listas (ou arrays), pondo o resultado na primeira lista.

A concatenação de vários arrays também é simples. Executando

```
@tar = (@aar, @bar, @car);
```

o array `@tar` fica com a concatenação dos três vectores nesta lista.

O *Perl* tem várias funções de manipulação de arrays e listas, algumas das quais já referimos. Vejamos as mais úteis: note que algumas são aplicáveis a strings ('listas' de caracteres).

- **chomp** `VARIABLE`
  **chomp** `LIST`
  **chomp**
  Veja a descrição de `chop` a seguir. `chomp` é semelhante a `chop`, sendo a única diferença o facto de remover o último caracter *apenas se fôr igual à variável pré-definida* `$/` (separador de registos no *Perl*) que por defeito é `'\n'`.

- **chop** `VARIABLE`
  **chop** `LIST`
  **chop**
  Esta função remove e devolve o último caracter da linha. Serve frequentemente para remover `'\n'` do fim das linhas. Se o argumento `VARIABLE` não fôr especificado, aplica-se implicitamente a `$_`. Quando aplicada a uma lista `LIST`, remove o último caracter de todas as strings na lista e devolve o caracter removido ao *último elemento* da lista.

- **die** `LIST`
  Termina a execução do programa e escreve na consola uma string que é a concatenação de `LIST` com a variável de erro interna `$!` (também conhecida por `ERRNO`).

- **defined** `EXPR`
  Esta função devolve um valor Booleano indicando se `EXPR` tem valor real ou não. Um escalar que não contém uma string, referência ou número válidos, é um valor *indefinido* ('undefined') ou, simplesmente, `undef`. Muitos operadores devolvem `undef` em condições de excepção (fim de ficheiro, erro de sistema, etc...). A função `defined` permite distinguir entre uma string nula indefinida e uma string nula definida.

  `defined` também permite avaliar da existência (ou definição) de arrays, hashes e subrotinas.

  Quando usado num elemento de um hash, `defined` apenas diz se o valor guardado nessa chave está ou não definido, nada diz sobre a chave em si. Para determinar se existe uma dada chave num hash usa-se a função `exists`.

- **pop** ARRAY
  **pop**
  Esta função usa um array como pilha ('stack'): remove e devolve o último elemento do ARRAY, encurtando-o. Se ARRAY fôr omitido, aplica-se a @ARGV (no programa principal) e a @_ dentro de subrotinas. O comportamento de pop é:

```
$tmp = pop @ARRAY;        # pop ...
$tmp = $ARRAY[$#ARRAY--]; # equivalência ...
$tmp = splice @ARRAY, -1; # outra equivalência
```

  Se o array está vazio, pop devolve undef. Veja também push, shift, unshift e splice (splice permite remover um maior número de elementos do array).

- **push** ARRAY, LISTA
  push usa também um array como stack: puxa os valores na LISTA para o fim do ARRAY e aumenta a dimensão deste de acordo com a dimensão da LISTA. A função devolve um escalar com a nova dimensão do ARRAY. Eis alternativas a push (note, porém, que push é mais eficiente):

```
push @ARRAY, @LISTA;
# equivalente a
foreach $value (LISTA) {
    $ARRAY[++$#ARRAY] = $value;
}
# equivalente ainda a
splice @ARRAY, @ARRAY, 0, @LISTA;
```

- **reverse** LISTA
  (Veja sort). Devolve uma lista com os elementos de LISTA por ordem inversa. É uma operação eficiente. Em contexto escalar, a função concatena todos os elementos da lista numa string e devolve-a pela ordem inversa, *caracter a caracter*.

- **shift** ARRAY
  **shift**
  Remove o primeiro valor do array e encurta-o de 1 elemento. Se ARRAY está vazio, shift retorna undef. Se ARRAY fôr omitido, shift aplica-se a @ARGV no programa principal e a @_ nas subrotinas. Veja também push, pop, unshift e splice. shift e unshift fazem, no início do ARRAY, o mesmo que pop e push fazem no fim.

- **sort** {code} LISTA
  com {code} omitido, ordena a LISTA por ordem crescente, segundo o critério de comparação alfabético, e devolve-a. {code} pode ser código anónimo ou o nome de uma subrotina. Veja o manual perlfunc para saber detalhes da escrita de code. O seguinte exemplo compara dois critérios de ordenação:

```
@w = (1,11,011);            # números 1, 11 e 9 (em octal)
@z = sort @w;               # ordenação (crescente) com critério alfabético
print "@z\n";               # imprime 1 11 9
@z = sort {$a <=> $b} @w;   # ordenação (crescente) com critério numérico
print "@z\n";               # imprime 1 9 11
```

- **splice** ARRAY, OFFSET, LENGTH, LISTA
  **splice** ARRAY, OFFSET, LENGTH

**splice** `ARRAY, OFFSET`
Esta função remove os elementos do `ARRAY` especificados pela posição inicial `OFFSET` e pela dimensão `LENGTH`, e substitui-os pela lista `LISTA`, se fôr especificada. A função devolve os elementos removidos. O array encurta ou cresce conforme necessário. Se `LENGTH` fôr omitido, a remoção é feita desde a posição `OFFSET` até ao fim do array. Se `OFFSET` fôr negativo, começa-se a contar do fim do `ARRAY`. Se o início do array `@a` fôr na posição 0 (o que é especificado pela variável especial `$[`) as seguintes equivalências são válidas:

```
push(@a, $x, $y);      # equivalente a splice(@a, $#a+1, 0, $x, $y);
pop(@a);               # equivalente a splice(@a, -1);
shift(@a);             # equivalente a splice(@a, 0, 1);
unshift(@a, $x, $y);   # equivalente a splice(@a, 0, 0, $x, $y);
$a[$x] = $y;           # equivalente a splice(@a, $x, 1, $y);
```

- **unshift** `ARRAY, LISTA`
  `unshift` faz no início do `ARRAY` o mesmo que `push` faz no seu fim: insere `LISTA` no início do `ARRAY` e devolve a sua nova dimensão (aumentada).

- **warn** `LISTA`
  Esta função escreve a mensagem contida em `LISTA` no handle `STDERR`,à semelhança de `die`, mas não termina o script nem acciona uma excepção.

### 3.4.1 Conversão entre 'arrays' e 'strings'

Um array é uma lista. Uma string pode ser uma lista de palavras (ou termos). O *Perl* tem funções para converter arrays em listas e vice-versa. Essas funções são:

- **join** `EXPR`, `LISTA`
  Concatena os elementos de `LISTA` numa única string, inserindo entre eles o(s) caracter(es) devolvido(s) pela expressão `EXPR`. Devolve a string assim construída. Por exemplo:

  ```
  @a = qw(a d f gh jk l);
  @b = qw/1 2 3 4 5/;
  $dd = join ':', @a, @b;    # a lista pode ser uma concatenação de listas
  print $dd,"\n";            # imprime 'a:d:f:gh:jk:l:1:2:3:4:5'
  ```

  Contrariamente a `split`, `join` não usa uma expressão regular no argumento `EXPR`.

  A forma mais eficiente de concatenar muitas strings é utilizar `join` com a string nula `''` no lugar de `EXPR` (é mais eficiente que o operador de concatenação `'.'`).

- **split** `/PATTERN/`, `EXPR`, `LIMITE`
  **split** `/PATTERN/`, `EXPR`
  **split** `/PATTERN/`
  **split**
  Esta função examina a string dada por `EXPR` para a existência dos delimitadores especificados por `/PATTERN/`, separa-a numa lista de substrings, e devolve essa lista

em contexto de lista ou o número de substrings em contexto escalar. O delimitador `PATTERN` pode ser uma expressão regular. Se o delimitador não fôr encontrado na string original, `split` devolve-a. Se `LIMITE` fôr especificado, o número de substrings não excede este valor. Se `EXPR` fôr omitida, `split` é aplicado implicitamente a `$_`. Se `/PATTERN/` fôr omitido, a separação é feita por espaços (i.e., o separador é a expressão regular `/\s+/`). Exemplos:

```
@caracteres = split //, $palavra;     # separa $palavra caracter a caracter
@campos = split /:/, $linha ;         # separa $linha pelos dois pontos ':'
@palavras = split ' ', $paragrafo ;   # separa $paragrafo por um espaço
($nome,$idade) = split / /, $registo, 2 ;   # lê primeiros 2 campos do $registo
```

No processamento de ficheiros de texto, é comum considerar cada linha como um registo. O *Perl* tem a variável `$/` que define o separador de registos e que vale `'\n'` por defeito.

## 3.5 Contexto escalar e contexto de lista

Um dos aspectos mais singulares do *Perl* é a distinção entre os contextos de lista e de escalar. O seguinte bloco de código ilustra a diferença:

```
@v = (1,2,3,4,5);
@u = @v;      # o array @u é uma cópia de @v
$a = @v;      # $a fica igual a 5 (i.e., à dimensão do array @v)
```

Observa-se pois, que um mesmo literal, `@v`, é interpretado de duas formas diferentes **consoante o contexto em que é interpretado**: num contexto escalar representa a dimensão do array; num contexto de lista representa o próprio array.

Note-se que não é apenas um literal array (começado por `@`) que pode ser avaliado de forma diferente nos dois contextos. A função `key %hash` devolve um array com todas as chaves do `%hash`. Veja, então, este exemplo

```
%hash = ('a',1,'b',2,'c',3,'d',4,'e',5,'f');    # 6 chaves, 5 elementos
@arr = keys %hash;        # contexto de lista
$numkeys = keys %hash;    # contexto escalar
print "@arr", "\n";
print "$numkeys", "\n";
```

imprime

```
e f a b c d
6
```

As chaves de `%hash` não são extraídas ordenadamente; se se pretendesse ordená-las quando são guardadas em `@arr` deveria fazer-se

```
@arr = sort keys %hash;
```

A função `sort` ordena a lista alfabeticamente (por ordem crescente). Se se pretender uma ordenação por ordem decrescente utiliza-se `reverse`

```
@arr = reverse sort keys %hash;
```

Se se quiser forçar uma lista a um contexto escalar pode usar-se

```
scalar(@list);
```

ou efectuar a adição escalar `@list + 0;`.

## 3.6 Programação funcional: `grep` e `map`

As funções `grep` e `map` permitem efectuar transformações em todos os elementos de uma lista. Este tipo de funções está associado a um paradigma de programação denominado *programação funcional*.

- **grep** `BLOCO LISTA`
  **grep** `EXPR, LISTA`
  `grep` executa o bloco de código `BLOCO`, ou a expressão `EXPR`, para cada elemento da `LISTA` (igualando, a nível local, `$_` a cada elemento). Em contexto de lista, devolve a lista de elementos para os quais `BLOCO` ou `EXPR` são verdadeiros. Em âmbito escalar, `grep` devolve o número de execuções 'verdadeiras'. Por exemplo, para remover linhas de comentário num ficheiro *Perl* (guardado no vector `@bar`), pode fazer-se
  ```
  @foo = grep(!/^\s*#/, @bar);
  ```
  ou, equivalentemente,
  ```
  @foo = grep {!/^\s*#/} @bar;
  ```
  `$_` pode ser usada para modificar os elementos do array (funciona como uma referência para cada elemento da lista). Esta modificação pode causar resultados bizarros se a lista não fôr um array nominal (e.g., se fôr uma lista anónima).

- **map** `BLOCO LISTA`
  **map** `EXPR, LISTA`
  `map` executa o bloco de código `BLOCO` ou a expressão `EXPR` para cada elemento da lista `LISTA` (igualando a nível local `$_` a cada elemento) e devolve a lista com os resultados da execução. Em contexto escalar, `map` devolve o número de elementos gerados.

  `BLOCO` ou EXPR são executados em contexto de lista, e assim cada elemento da `LISTA` pode gerar zero, um, ou mais elementos na lista devolvida. Por exemplo
  ```
  @chars = map(chr, @nums);
  ```
  traduz uma lista `@num` contendo números para os respectivos caracteres ASCII e
  ```
  %hash = map { getkey($_) => $_ } @array;
  ```
  é uma maneira 'curtida' de fazer

  ```
  %hash = ();
  foreach $_ (@array) {
      $hash{getkey($_)} = $_;
  }
  ```

  Mais uma vez se chama a atenção para a modificação da `LISTA` através da referência implícita `$_`: pode causar resultados estranhos se a `LISTA` não fôr nominal.

A programação funcional permite 'compactar' eficazmente operações em listas de elementos. Por exemplo, considere a lista de números `@n = (1,2,3,5,6,7,9)` cujos cubos quer imprimir. Isto é feito compactamente com
```
print map($_**3 , @n);
```
que imprime `1827125216343729`, o que não é possivelmente o que pretende, pois os números são escritos sem separador. Para os imprimir separados por um espaço, modifica-se a anterior instrução por forma a juntar um espaço a cada cubo:

```
print map($_**3 . ' ' , @n), "\n";
```
que escreve `1 8 27 125 216 343 729` na consola.

## 3.7 'Hashes' (dicionários)

Os hashes, ou dicionários, são listas de elementos indexadas por uma **chave** do tipo **string,** contrariamente aos arrays que são indexados por um inteiro positivo. Os hashes são inicializados da seguinte forma:

```
@hax = ('a', 10, 'bb', "hello", 20, 'Ist');
@hax = ('a' => 10, 'bb' => "hello", 20 => 'Ist');    # equivalente
```
Esta última forma é mais legível. A seta `'=>'` é equivalente à vírgula na definição de listas (também podia ser utilizada a forma alternativa de criação de listas `qw/.../`).

O *Perl* tem funções dedicadas à manipulação e processamento de hashes.

- **dbmclose** `%HASH`
  Quebra a ligação estabelecida entre uma base de dados (tipo DBM) e o `%HASH`. Veja `dbmopen`.

- **dbmopen** `%HASH, dbname, mode`
  Estabelece ligação entre uma base de dados (BD) `dbname` (em formato DBM) e o `%HASH`. O modo `mode` é aquele utilizado em Unix para especificar as permissões associadas a ficheiros (e.g. 0777 permite a leitura e a escrita do ficheiro a todos os utilizadores). A execução de `dbmopen` cria os ficheiros `dbname.pag` e `dbname.dir`, caso a BD ainda não tenha sido criada. `dbname` pode incluir um 'path' (e.g, `c:/bdados/dbname)`. Apenas os valores inseridos em `%HASH` depois da execução de `dbmopen`, ou já existentes na BD, estão acessíveis. Eis um exemplo de inicialização de uma BD (são criados na directoria de trabalho os ficheiros 'xpto.pag' e 'xpto.dir'):

  ```
  dbmopen %xpto, 'xpto', 0777;
  $xpto{'a'} = 100 ;
  dbmclose %xpto;
  ```

  `dbmopen` e `dbmclose` podem ser utilizadas para realizar BDs de média complexidade.

- **delete** `$HASH{HASHKEY}`
  Apaga a chave `$HASH{HASHKEY}` do `HASH`. O argumento de `delete` pode ser qualquer expressão arbitrária que devolva a chave de um elemento de `HASH`.

- **each** `HASH`
  Esta expressão devolve uma lista de 2 elementos com a `CHAVE` e o respectivo `VALOR` no hash `HASH`. Utilizando sucessivamente `each`, pode examinar-se completamente um hash. Após o hash ser completamente lido, `each` devolve uma lista nula que é falsa num teste condicional. Por exemplo

  ```
  while (($key,$value) = each %hash) { print "$key,$value\n"; }
  # escreve todos os pares chave-valor existentes no %hash
  ```

- **exists** `$HASH{HASHKEY}`
  Esta função retorna 'verdade' se a chave `HASHKEY` existe no hash `HASH`, mesmo que o

seu valor seja `undef`. O argumento de `exists` pode ser qualquer expressão ou função que devolva uma chave de hash.

- **keys** `HASH`
  Devolve uma lista com todas as chaves do hash `HASH`, numa ordem aparentemente aleatória. Esta ordem é a mesma encontrada na aplicação das funções `each` ou `values` ao hash (se este não foi modificado entretanto).

  Em contexto escalar, `keys` dá o número de elementos do hash (e faz 'reset' ao iterador `each`). `keys` pode ser utilizada numa atribuição, permitindo aumentar a memória utilizada pelo hash (o que melhora o tempo de acesso)
  ```
  keys %HASH = 200;
  ```

- **values** `HASH`
  Devolve uma lista com todos os valores guardados no hash `HASH`, pela mesma ordem que seria obtida na aplicação das funções `each` ou `keys` (se aquele entretanto não foi modificado).

## 3.8 Variáveis especiais

O *Perl* implementa um grande número de variáveis pré-definidas. Algumas são utilizadas implicitamente em diversas funções. Permitem escrever rapidamente código denso e compacto (pois não são visíveis no código as variáveis envolvidas). Como diz Larry Wall, o pai e mãe do Perl, "todo o programador é preguiçoso".

Lembre-se que no C existem duas variáveis pré-definidas que permitem aceder aos argumentos da linha de comando: `argc` e `argv`. Pode pensar, se quiser, que o *Perl* tem uma grande variedade (e quantidade...) destas variáveis especiais.

A maioria destas variáveis tem um 'nick-name' constituído pelo caracter associado ao seu tipo (`$` para um escalar, `@` para um array e `%` para um hash) e por outro (ou outros) caracter não alfanumérico. Exemplos: `$_, @_, @ARGV, %ENV, $[, ....`

Para além do 'nick-name' as variáveis pré-definidas têm uma denominação respeitável utilizada pelos programadores 'com classe' `8-)`. Para invocar estes nomes respeitáveis escreve-se o pragma `use English;` no início do programa. Algumas variáveis têm 3 denominações, ao todo, sendo a terceira 'emprestada' pela linguagem de programação `awk`.

Na secção do manual denominada `perlvar` encontra-se uma lista completa destas variáveis (faça `perldoc perlvar` para a consultar). Eis aqui uma selecção das mais importantes (as denominações 'compridas' estão associadas à invocação do pragma `use English;`):

- **$ARG**, **$_**
  É o escalar utilizado por defeito para entrada de dados. Os seguintes pares de instruções são equivalentes:

  ```
  while (<>) {...}      # equivalente só num while!
  while (defined($_ = <>)) {...}

  /^Subject:/           # coincidência (acerto) em expressões regulares
  $_ =~ /^Subject:/

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

```
$_ =~ tr/a-z/A-Z/;

chomp;                # remoção de \n no fim de $_
chomp($_);
```

O *Perl* assume implicitamente `$_` como argumento em várias funções, quando estas são invocadas sem argumento:

- Várias funções unárias, incluindo `ord()` e `int()`, assim como todos os testes a ficheiros (`-f`, `-d`, ...) excepto `-t`, que testa `STDIN` por defeito.
- Várias funções aplicáveis a listas, `print()` e `unlink()`.
- Os operadores de 'pattern matching' como `m//`, `s///`, e `tr///`.
- `$_` é a variável de iteração por defeito em listas sujeitas ao ciclo `foreach` (ou `for`) se não fôr especificada qualquer variável de iteração:
  ```
  foreach (@lista) { processa... }     # $_ itera a @lista
  for (@lista) { processa... }         # equivalente
  ```
- `$_` é a variável de iteração implícita nas funções `grep()` e `map()`.
- `$_` é a variável para onde é lido um registo de entrada quando um operador `<FH>` (leitura de filehandle) é testado como único critério num `while`. Fora dum teste a um `while` isto não acontece.

- **$<digits>**
  (Por exemplo `$1`, `$2` ...) Contêm a 'sub-pattern' no correspondente par de parêntesis curvos `(...)` após uma coincidência bem sucedida com expressões regulares. As variáveis `$<digits>` apenas podem ser lidas, são locais ao bloco onde a coincidência está a ser feita e são criadas dinamicamente. Veja a secção dedicada às expressões regulares ou o manual `perlre`.

- **$MATCH**, **$&**
  A string resultante da última pesquisa de texto ('pattern match') bem sucedida. Esta variável apenas pode ser lida, é local ao bloco onde a pesquisa está a ser feita e é dinâmica. (A sua utilização penaliza fortemente o tempo de execução das operações com expressões regulares).

- **$PREMATCH**, **$‘**
  **$POSTMATCH**, **$’**
  **$LAST_PAREN_MATCH**, **$+**
  **@LAST_MATCH_END**, **@+**
  **@LAST_MATCH_START**, **@-**
  **$MULTILINE_MATCHING**, **$*** (o uso desta variável é desaconselhado)
  Estas variáveis têm a ver com o sucesso numa pesquisa de texto com expresssões regulares. Consulte o manual `perlre` para detalhes.

- **$INPUT_LINE_NUMBER**, **$NR**, **$.**
  O número correspondente ao corrente registo ('record') do último filehandle lido com `read()`, `seek()`, ou `tell()`. Habitualmente, um registo é uma linha e o fim do registo é assinalado por `'\n'` (para mudar o caracter que assinala o fim do registo altera-se `$/`). Um `close` explícito do ficheiro faz `$.` igual a zero. Como `<>` não

efectua um `close` explícito, `$.` vai sempre aumentando quando se lê uma lista de ficheiros através de `@ARGV` com o operador `<>`.

- **$INPUT_RECORD_SEPARATOR**, **$RS**, **$/**
  O separador de registos de entrada. Por defeito, é o fim-de-linha `'\n'`. Esta variável 'altera a ideia' do *Perl* sobre o fim de linha. Considera as linhas em branco como o separador de registos quando se faz `$/=""` (i.e., é a string nula). Se `$/` fôr igualada a um conjunto de caracteres, será este conjunto que termina cada registo (e.g., `$/="-:-"`). Se fôr aplicada a função `undef` a `$/`, a leitura do ficheiro com uma variável escalar 'engole' ('slurp') todo o ficheiro. Veja este exemplo

```
undef $/;           # entra em modo de 'engolir'
$_ = <FH>;          # lê todo o ficheiro
s/\n[ \t]+/ /g;     # substitui tabs e newlines por espaços
```

  Atenção: o valor que se atribui a `$/` é uma string e não uma regex.

  Igualando `$/` a uma referência para inteiro, para escalar contendo um inteiro, ou para escalar convertível para inteiro, faz-se com que a leitura seja realizada por registos (e não por linhas) com uma dimensão máxima definida por aquele inteiro. Assim

```
$/ = \32768;    # ou \"32768", ou \$var_contendo_32768
open(FILE, $myfile);
$_ = <FILE>;
```

  lê um registo com não mais do que 32768 bytes do ficheiro `$myfile`.

- **$OUTPUT_AUTOFLUSH**, **$|**
  Se diferente de zero, força o `flush` (escrita) do 'buffer' de saída de dados após cada `write` ou `print`. O valor por defeito é 0.

- **$OUTPUT_FIELD_SEPARATOR**, **$OFS**, **$,**
  O separador dos campos no operador `print` (neste operador os campos são separados por ',' e assim `$,` define os caracteres escritos em vez da vírgula). Pode ser alterada pelo programador.

- **$OUTPUT_RECORD_SEPARATOR**, **$ORS**, `$\`
  O separador de cada registo no operador `print`, ou seja, o que é escrito no canal de saída (e.g., na consola) no fim de cada linha escrita com `print`.Pode ser alterada pelo programador: por exemplo, se se fizer `$\="\n";` muda-se de linha com cada `print` (`$\=''` por defeito).

- **$LIST_SEPARATOR**, **$"**
  Esta variável funciona como `$,` só que se aplica na escrita de arrays (ou fatias de arrays) interpolados numa string. Por defeito é um espaço.

- **$SUBSCRIPT_SEPARATOR**, **$SUBSEP**, **$;**
  O separador de índices para emulação de arrays multidimensionais com hashes. Se se referir um elemento de um hash com `$foo{$a,$b,$c}` o *Perl* transcreve isto para `$foo{join($;, $a, $b, $c)}`. Atenção, não use `@foo{$a,$b,$c}` que representa

uma fatia ('slice') pois começa por `@` e significa `($foo{$a},$foo{$b},$foo{$c})`. O valor por defeito de `$;` é "\034". Na prática, deve utilizar-se arrays multidimensionais "reais"como é descrito no manual `perllol`.

- **$OFMT**, **$#**
  O formato de impressão de números. O valor de defeito é `"%.ng"`. A modificação de `$#` é desaconselhada.

- **$FORMAT_PAGE_NUMBER**, **$%**
  (e ainda `$= $- $~ $^ $: $^L $^A`). Variáveis utilizadas em 'formatos' (uma facilidade disponibilizada pelo *Perl* para formatar relatórios. Consulte a página `perlform` do manual.)

- **$CHILD_ERROR**, **$?**
  Variável que armazena o estado da última 'chamada' ao sistema operativo.

- **$OS_ERROR**, **$ERRNO**, **$!**
  Se utilizado como string, dá a string de erro de sistema correspondente.

- **$EXTENDED_OS_ERROR**, `$^E`
  Erro de sistema específico em alguns sistemas operativos. Na maioria dos sistemas operativos é igual a `$!`.

- **$EVAL_ERROR**, **$@**
  A mensagem de erro de sintaxe após a última execução de um `eval()`. Se fôr a string nula (i.e. igual a `""`) o último `eval()` encontrou uma sintaxe correcta e executou o código sem erros.

- **$PROCESS_ID**, **$PID**, **$$**
  O número do processo correspondente à execução do presente script pelo *Perl* (considere a variável só de leitura).

- **$REAL_USER_ID**, **$UID**, **$<**
  **$EFFECTIVE_USER_ID**, **$EUID**, **$>**
  **$REAL_GROUP_ID**, **$GID**, **$(**
  **$EFFECTIVE_GROUP_ID**, **$EGID**, **$)**
  Variáveis relacionadas com a identificação do utilizador que executa o *Perl* e que permite controlar os seus privilégios. São importantes para a segurança do sistema.

- **$PROGRAM_NAME**, **$0**
  O nome do programa (script) que está em execução.

- **$[**
  Especifica qual o índice do primeiro elemento de um array e do primeiro caracter duma string. Vale 0 por defeito. Pode ser igualada a 1 (por exemplo), mas é desaconselhada a sua alteração.

- **$]**
  Contém a "versão + 'patchlevel' / 1000"do interpretador *Perl*. Pode ser utilizada para determinar se se tem a versão adequada de *Perl* para executar um dado script. Veja também a documentação de `use VERSION;` e de `require VERSION;` para saber como terminar a interpretação se o *Perl* estiver desactualizado. O uso desta variável é desaconselhado.

- **$OSNAME**, `$^O`
  O nome do sistema operativo no qual a actual cópia do *Perl* foi compilada, conforme determinado na configuração de compilação. Veja o interruptor `-V` no manual `perlrun`.

- **$EXCEPTIONS_BEING_CAUGHT**, `$^S`
  Actual estado de execução do interpretador. Não definido se o 'parsing' do actual `module/eval` não foi completado. Verdadeiro dentro de um bloco `eval()`, falso fora dele.

- **$BASETIME**, `$^T`
  O instante de tempo em que o programa começou a execução, em segundos, desde a época (início de 1970). Os valores devolvidos pelos testes a ficheiros `-M`, `-A`, e `-C` são baseados neste valor.

- **$PERL_VERSION**, `$^V`
  Informação sobre a versão, a subversão e a revisão do *Perl* que está a ser usada.

- **$WARNING**, `$^W`
  O actual valor do interruptor de avisos (`-w`). É `true` ou `false` consoante foi usado ou não no comando de invocação do script. Pode ser modificado no script.

- **$EXECUTABLE_NAME**, `$^X`
  O nome do executável Perl.

- **$ARGV**
  Contém o nome do actual ficheiro em processamento, quando se lê argumentos da linha de comando utilizando `<>`.

- **@ARGV**
  O array `@ARGV` contém os argumentos fornecidos na linha de comando que invocou o script. `$#ARGV` é igual ao número de argumentos menos um, porque `$ARGV[0]` é o primeiro argumento, não o nome do programa. `$0` contém o nome do programa.

- **@INC**
  O array `@INC` contém a lista de directorias em que os comandos `do EXPR`, `require`, e `use` procuram módulos. Inicialmente, consiste dos argumentos especificados com os interruptores `-I` e da localização da biblioteca de módulos 'nativa' do *Perl* (e.g. `/usr/local/lib/perl` seguido de `'.'`, a actual directoria). Se fôr necessário alterar `@INC` dentro do script, deverá usar-se o pragma `use lib` para carregar adequadamente bibliotecas dependentes da máquina:

  ```
  use lib '/mypath/libdir/';
  use SomeMod;
  ```

- **@_**
  Dentro de uma subrotina (ou subprograma), o array `@_` contém os parâmetros passados para essa subrotina. Consulte a página de manual `perlsub` e a secção deste documento, mais adiante, em que se discutem os subprogramas.

- **%INC**
  O hash `%INC` contém os ficheiros incluídos com os operadores `do`, `require`, ou `use`. A chave é o nome do ficheiro especificado (com os nomes dos módulos convertidos para o respectivo 'path') e o valor guardado no hash `%INC` indica a localização do ficheiro. O operador `require` usa este hash para determinar se um ficheiro já foi incluído.

- **%ENV**, **$ENV{expr}**
  O hash `%ENV` contém as actuais variáveis de sistema. Alterando um valor em `%ENV` altera-se o ambiente de execução para qualquer processo 'filho' do actual script.

# 4 Referências: princípios básicos

(Nota: o tratamento das referências nesta secção segue de perto os manuais `perlref` e `perllol` ).

Em *Perl5* é possível utilizar referências simbólicas e referências fortes ('hard'). As referências são variáveis escalares. Utilizando arrays e hashes preenchidos com referências é possível construir estruturas de dados complexas como arrays de arrays, arrays de hashes, hashes de arrays, arrays de hashes de funções, etc. Nesta secção vai discutir-se este tópico, pois estas estruturas são fundamentais em programas que manipulam dados com relações complexas entre si.

As referências fortes (obtidas pela prefixação de `'\'` ao ítem que vão referir) contabilizam o número de vezes que a estrutura é referida e dereferida, e fazem 'recolha de lixo' ('garbage collection') automática, libertando para posterior utilização a memória correspondente às estruturas que deixam de ser utilizadas.

As referências simbólicas são apenas strings com nomes de variáveis ou de outros objectos. Na prática, em *Perl5* usa-se quase sempre referências fortes.

É fácil usar referências em *Perl* e há apenas uma regra que deve ser sempre seguida: o *Perl* não derefere implicitamente. Se um escalar contém uma referência, comporta-se como um simples escalar e não se torna, magicamente, no array, hash ou subrotina que refere. Ele tem que ser de-referido explicitamente para se atingir o objecto apontado.

## 4.1 Criação de referências

Criam-se referências através de vários métodos.

1. Usando o operador `'\'` numa variável, subrotina ou valor. Eis exemplos:

   ```
   $escalarref = \$foo;
   $arrayref  = \@ARGV;
   $hashref   = \%ENV;
   $coderef   = \&handler;
   ```

   Referências para filehandles são mais complexas de realizar, embora isso possa ser feito.

2. Uma referência para um array anónimo pode ser criada com parêntesis rectos:
   `$arrayref = [1, 2, 3, ['a', 'b', 'c']];`
   Criou-se uma referência para um array anónimo de quatro elementos, cujo último elemento é também uma referência para um array anónimo com três elementos. A sintaxe de acesso a este array multidimensional anónimo é exemplificada por `$arrayref->[2][1]` que tem o valor `'b'`.

   Referir uma lista enumerada não é o mesmo que usar parêntesis rectos – pelo contrário, corresponde a criar uma lista de referências!

   ```
   @lista = (\$a, \@b, \%c);
   @lista = \($a, @b, %c);      # o mesmo
   ```

Um caso especial é `\(@foo)` ou `\(%foo)`, que cria uma lista com referências para cada elemento de `@foo` ou de `%foo` e não uma referência para `@foo` ou `&foo`.

3. Uma referência para um hash anónimo é criada com chavetas.

```
$hashref = {
    'Adam'  => 'Eve',
    'Clyde' => 'Bonnie',
};
```

Pode combinar-se arrays e hashes anónimos para criar estruturas de dados complexas. A sintaxe multidimensional funciona sempre. O anterior exemplo utilizou literais, mas pode utilizar-se variáveis e expressões como elementos dos hashes. Como as chavetas são usadas para delimitar blocos, por vezes é necessário indicar explicitamente ao *Perl* que a chaveta não marca o início de um bloco, antecedendo-a de `+` ou de `return`. Por exemplo, se se pretendesse escrever uma função que criasse um novo hash e devolvesse a respectiva referência, tínhamos 3 opções (uma delas errada):

```
sub rethashref {        { @_ } }   # errada: o perl vê um bloco...
sub rethashref {       +{ @_ } }   # ok: +{ desambigua
sub rethashref { return { @_ } }   # ok: return desambigua
```

Se se pretender, de facto, um bloco, pode-se fazer:

```
sub retarray {        { @_ } }   # ambíguo (actualmente ok, mas...)
sub retarray {       {; @_ } }   # ok
sub retarray { { return @_ } }   # ok
```

Os termos `+{` e `{;` servem para remover a ambiguidade na interpretação da chaveta, relativamente a ser considerada como uma referência para hash ou apenas como delimitadora de um bloco.

4. Uma referência para uma subrotina anónima é criada utilizando o termo `sub` sem atribuir nome à subrotina:
   `$coderef = sub { print "Boink!\n" };`
   Note que tem que ser colocado um ponto-e-vírgula. Nesta criação o código da subrotina não é executado.

5. As referências podem ser devolvidas por subrotinas especiais denominadas *construtores*. Esta técnica de criação de referências relaciona-se com OOP e não é aqui discutida.

6. Uma referência pode ser criada com uma sintaxe especial, carinhosamente denominada sintaxe **'*foo{THING}'**. `*foo{THING}` devolve uma referência para o tipo THING em `*foo` (que é a denominação do campo da tabela de símbolos do *Perl* que engloba todos os nomes `foo`: `@foo`, `$foo`, `%foo`...)

```
$scalarref = *foo{SCALAR};
$arrayref  = *ARGV{ARRAY};
$hashref   = *ENV{HASH};
$coderef   = *handler{CODE};
$ioref     = *STDIN{IO};
$globref   = *foo{GLOB};
```

`*STDIN{IO}` relaciona-se com o acesso a entidades exteriores (filehandles, dirhandles, sockets).

`*foo{THING}` devolve `undef` se a THING em questão ainda não foi utilizada, excepto no caso de escalares pois `*foo{SCALAR}` devolve uma referência para um escalar anónimo se `$foo` ainda não existe.

## 4.2 Utilização de referências

Há vários métodos de utilização de referências.

1. Onde é possível colocar um identificador (ou cadeia de identificadores), pode-se utilizar uma variável escalar contendo uma referência para o tipo correcto:

   ```
   $bar = $$escalarref;
   push(@$arrayref, $filename);
   $$arrayref[0] = "Janeiro";
   $$hashref{"KEY"} = "VALOR";
   &$coderef(1,2,3);
   print $globref "output\n";
   ```

   É importante compreender que nestes exemplos não se está a dereferir `$arrayref[0]` ou `$hashref{"KEY"}`. A dereferência da variável escalar acontece antes do acesso aos elementos dos arrays ou hashes. Referências mais complexas, deverão utilizar os métodos 2 e 3 que se seguem.

2. Um identificador pode-se ser substituído por um `BLOCO` de código que devolve uma referência do tipo correcto. Por outras palavras, os anteriores exemplos poderiam ser escritos como:

   ```
   $bar = ${$escalarref};
   push(@{$arrayref}, $filename);
   ${$arrayref}[0] = "Janeiro";
   ${$hashref}{"KEY"} = "VALOR";
   &{$coderef}(1,2,3);
   $globref->print("output\n");  # se se utilizar o módulo IO::Handle
   ```

   Nestes exemplos simples é exagerado utilizar `{...}`, mas o `BLOCO` pode conter expressões arbitrárias, em particular expressões indexadas como, por exemplo:

   ```
   &{ $dispatch{$index} }(1,2,3);      # chama a subrotina  correcta
   ```

   A omissão de `{...}` nos casos simples não causa confusões. Veja, porém, os seguintes casos, onde o caso 0 é uma simplificação do caso 1, não do caso 2:

```
$$hashref{"KEY"}   = "VALOR";       # CASO 0
${$hashref}{"KEY"} = "VALOR";       # CASO 1
${$hashref{"KEY"}} = "VALOR";       # CASO 2
${$hashref->{"KEY"}} = "VALOR";     # CASO 3
```

O caso 2 é enganador na medida em que se acede ao hash denominado `%hashref`: não se está a dereferir a referência `$hashref` e a aceder ao hash que ela aponta. Isto acontece no caso 3.

3. A chamada de subrotinas e o acesso a elementos de arrays e de hashes é tão frequente (e pode causar tantas confusões) que o *Perl* disponibiliza uma forma sintáctica 'açucarada' para ser utilizada no anterior caso 2. A de-referência pode ser implementada com uma **seta** `'->'`, como se mostra de seguida:

```
$arrayref->[0] = "Janeiro";   # Elemento de array
$hashref->{"KEY"} = "VALOR";  # Elemento de hash
$coderef->(1,2,3);            # Invocação de subrotina
```

O lado esquerdo do operador `'->'` pode ser uma expressão que devolve uma referência, incluindo uma de-referência prévia. Note que `$array[$x]` não é o mesmo que `$array->[$x]`:

```
$array[$x]->{"foo"}->[0] = "Janeiro";
```

Este é um dos casos em que referências podem ser criadas num contexto de 'lvalue' ('left value', i.e., aparecem à esquerda de um operador). Antes desta instrução, `$array[$x]` poderia estar indefinido. Se assim fosse, seria automaticamente criado e preenchido com uma referência para um hash para que se pudesse procurar `{"foo"}` nele. Da mesma forma, `$array[$x]->{"foo"}` seria automaticamente definido como uma referência para array de forma a ser possível procurar `[0]`. Este processo é chamado *auto-vivificação*.

O operador `->` é de utilização opcional entre parêntesis contendo índices. Assim, pode compactar-se a anterior instrução para

```
$array[$x]{"foo"}[0] = "Janeiro";
```

Quando se usam apenas arrays, esta sintaxe permite referir arrays multidimensionais como é feito em em C:

```
$score[$x][$y][$z] += 42;
```

Bem, OK, não inteiramente como em C. O C não sabe aumentar automaticamente um array quando isso lhe é pedido, e o *Perl* sabe `8-)`.

A utilização de uma string ou de um número como referência, cria uma referência simbólica. A utilização de uma referência como se de número se tratasse, devolve um número que representa a posição de memória apontada. A única tarefa útil que pode ser realizada com este comportamento, é a comparação numérica entre referências, para saber se apontam para uma mesma localização.

```
if ($ref1 == $ref2) {      # comparação numérica de referências
    print "as refs 1 e 2 referem o mesmo objecto\n";
}
```

A utilização de uma referência em contexto de string, devolve o seu tipo e o respectivo endereço numérico em caracteres hexadecimais. O operador `ref($umaref)` devolve apenas o tipo apontado pela referência `$umaref` sem o endereço.

Um 'typeglob' pode ser dereferido da mesma forma que uma referência, porque a sintaxe de dereferência indica sempre o tipo de referência desejada. Assim, `${*foo}` e `${\$foo}` indicam ambas a mesma variável escalar `$foo`.

Eis um truque para interpolar uma chamada de subrotina numa string:

```
print "A sub devolveu @{[mysub(1,2,3)]} agora.\n";
```

Isto funciona do seguinte modo: quando `@{...}` é encontrado na string entre aspas (interpolada), é executado como um bloco. O bloco cria uma referência para um array anónimo contendo o resultado da chamada a `mysub(1,2,3)`. Assim, o bloco global devolve uma referência para um array, inserido na string, que é dereferido por `@{...}`. Esta 'aldrabice' é também útil para embeber expressões arbitrárias em strings:

```
print "Chegaram @{[$n + 5]} bruxas\n";
```

## 4.3 Referências Simbólicas

Disse-se que as referências são criadas quando necessário, se não estiverem ainda definidas, mas não se disse o que acontece se um valor utilizado como referência já está definido mas não é, de facto, uma referência forte ('hard'). Ele irá ser tratado como uma referência simbólica: o valor do escalar é assumido como um nome de variável:

```
$name = "foo";
$$name = 1;                # Atribui 1 a $foo
${$name} = 2;              # Atribui 2 a $foo
${$name x 2} = 3;          # Atribui 3 a $foofoo
$name->[0] = 4;            # Atribui 4 a $foo[0]
@$name = ();               # Esvazia @foo
&$name();                  # Chama &foo() (como se fazia no Perl 4)
```

Esta sintaxe é poderosa, mas perigosa, pois é possível pretender utilizar-se uma referência forte e, acidentalmente, implementar uma referência simbólica. Para prevenir estes erros pode ser invocado o pragma

```
use strict 'refs';
```

e, assim, somente referências fortes serão admitidas no actual bloco. Num outro bloco poderá ser alterarada esta restrição com o pragma

```
no strict 'refs';
```

Somente as variáveis globais são visíveis com referências simbólicas. As variáveis léxicas (declaradas com `my()`) não entram na tabela de símbolos e, por isso, são invisíveis a este mecanismo. Por exemplo:

```
local $value = 10;
$ref = "value";
{
    my $value = 20;
    print $$ref;
```

```
}
```

imprime 10 e não 20, pois a declaração da variável `$value` com `local()` funciona como declaração de variável global neste caso.

# 5 Processamento condicional e iterativo

O *Perl* possui operadores de selecção (processamento condicional) e de iteração. Antes de passar à sua análise, vejamos o que é e o que não é **verdadeiro** em *Perl*.

- Qualquer string é verdadeira, excepto `""` e `"0"`.
- Qualquer número é verdadeiro excepto `0`.
- Qualquer referência é verdadeira.
- Qualquer valor indefinido é falso.

Há comportamentos aparentemente 'estranhos' na aplicação destas regras. Por exemplo, o número `0.00` é considerado falso, mas a string `"0.00"` é considerada verdadeira (é uma *string* diferente de `"0"`). Porém, `"0.00"+0` é uma operação que tem como resultado o escalar `0` que é considerado falso: assim, `"0.00"+0` é uma expressão cujo teste lógico é falso.

## 5.1 Selecção e processamento condicional

O *Perl* possui a estrutura de selecção condicional `if` {...} `elsif` {...} ... `elsif` {...} `else` {...} mas não suporta a construção `case` existente noutras linguagens. O número de cláusulas `elsif` não tem limite. Vejamos um exemplo.

```
if ($idade < 0) {
  print "Erro!\n";
} elsif ($idade < 3) {
  print "Bébé!\n";
} elsif ($idade > 70) {
  print "Boa, avô!\n";
} else {
  print "Pague o imposto!\n";
}
```

Existe a instrução `unless` que é o complemento lógico de `if`. As instruções que se seguem são equivalentes:

```
if ($idade < 0) {
  print "Erro!\n";
}
# é equivalente a
unless ($idade >= 0) {
  print "Erro!\n";
}
```

Ou seja, `if (condition) {...}` equivale a `unless (not condition) {...}`.

A construção `unless` não tem cláusulas equivalentes ao `else` e ao `elsif`.

## 5.2 Iteração

As estruturas de iteração em *Perl* são muito usadas no processamento de listas. Começa-se pelo `while`, que habitualmente é utilizado da seguinte forma:

```
while (COND) {
  ......
  # algures, numa execução deste bloco, COND torna-se falsa
  ......
}
# continua aqui, após COND ser falsa
```

Quando se processa o `while`, a condição `COND` é avaliada: se fôr falsa, o programa continua a execução após o bloco `while`; se fôr verdadeira, o bloco `while` (entre o par de chavetas) é executado até ao fim. No final de cada execução, volta-se ao início e testa-se a condição `COND`: se fôr verdadeira executa-se de novo o bloco; se fôr falsa, sai-se do bloco e a execução continua abaixo.

Pode alterar-se a sequência de execução dentro do bloco `while` usando `next`, `redo` e `last` (veja a utilização equivalente destas instruções numa iteração `for`, mais abaixo).

O `while` pode ser seguido, opcionalmente, de um bloco `continue`,

```
while (COND) {
  ......
  # algures, numa execução deste bloco, COND torna-se falsa
  ......
} continue {
  ......
}
```

O bloco `continue` é sempre executado, mesmo se a sequência de processamento dentro do `while` fôr alterada devido, por exemplo, a uma instrução `next`. O `continue` é raramente utilizado.

O `while` serve para processar listas e ficheiros. Por exemplo, o código

```
  while ($line = <CIRCUITO>) {
      # processa $line...
  }
```

lê, em cada execução do bloco interior ao `while`, uma linha do ficheiro apontado pelo filehandle `CIRCUITO` e processa-a, até ser atingido o fim do ficheiro, onde a condição `($line = <CIRCUITO>)` é falsa e o `while` termina. Outro exemplo é:

```
  while (@ARGV) {
      # processa(shift @ARGV);
  }
```

em que se processa os elementos de `@ARGV` até este ficar vazio (`shift` remove e devolve o primeiro elemento de `@ARGV` cada vez que é executado). Em `'while (@ARGV)'` o array é

avaliado em contexto escalar e devolve o número de elementos (é equivalente a `'while (scalar(@ARGV))'`). Assim, esta condição é falsa se `@ARGV` estiver vazio.

Existe também a instrução `until` que funciona de forma análoga ao `while` excepto no pormenor de que o bloco só é executado se a condição de teste fôr falsa:

```
while (COND) {
    ....
}
# é equivalente a
until (!COND) {        # ! é o operador de negação
    ....
}
```

Neste exemplo

```
$c = 10;
while ($c--) { print $c, ' '; }
print "\n";
$c = 10;
until (!$c--) { print $c, ' '; }
print "\n";
```

ambos os ciclos `while` e `until` imprimem `9 8 7 6 5 4 3 2 1 0`.

A instrução `for` em *Perl* é bastante semelhante àquela disponibilizada em C. As instruções de controle `next`, `last` e `redo` permitem alterar a sequência de processamento dentro de ciclos `for`. Eis o protótipo da sua utilização:

```
for (EXPRINIT ; COND ; EXPRMOD) {
  # CONDREDO transfere a execução para aqui, o início do bloco, sem ser testada COND
  .......
  next if CONDNEXT ;
  .......
  last if CONDLAST ;
  .......
  redo if CONDREDO;
  .......
  # CONDNEXT transfere a execução para aqui, o fim do bloco...
}
# CONDLAST termina o bloco, e o programa continua aqui...
```

O ciclo `for` faz uso de três expressões. No começo do ciclo, executa a expressão inicial `EXPRINIT` e verifica se a condição `COND` é verdadeira. Se fôr, executa o corpo do ciclo (entre chavetas) até ao fim, após o que executa a expressão `EXPRMOD` que modifica a variável de ciclo testada em `COND`. De seguida, testa `COND` e consoante esta seja verdadeira ou falsa, assim repete ou não a execução do bloco.

Na execução do bloco, se `CONDNEXT` fôr verdadeira a execução passa para o fim do bloco, 'curto-circuitando' todas as instruções após `next if CONDNEXT;`. Se `CONDLAST` fôr verdadeira, a execução é transferida para fora do bloco e o programa continua a ser executado a partir daí. Finalmente, se `CONDREDO` fôr verdadeira, transfere-se a execução

para o código que se segue à instrução `for` sem testar a condição `COND`. Quando usadas dentro de uma estrutura iterativa `while`, o comportamento destas instruções de controle é semelhante. A ordem pela qual aparecem `next`, `last` e `redo` no bloco é livre.

A instrução `continue` associada ao `while` permite fazer uma equivalência 'exacta' entre um `for` e um `while`, como se ilustra de seguida:

```
for ($i = 1; $i < 10; $i++) {
  ......
}
# equivale a
$i = 1;
while ($i < 10) {
  ......
} continue {
  $i++;
}
```

Sem a `continue` não seria possível a equivalência 'exacta' entre o `for` e o `while`.

Para sair de dentro de vários ciclos encadeados, usam-se `LABELS`. Veja o exemplo:

```
......
LABELFOR: for ($i=0; $i<= $#myarr; $i++) {
  ......
  $val = $myarr[$i];
  ......
  LABELWHILE: while ($val !~ /x/) {
      ......
      next LABELFOR if COND1 ;
      ......
      last if COND2 ;         # ou last LABELWHILE if COND2
      ......
      last LABELFOR if COND3 ;
      ......
      next if COND4 ;         # ou next LABELWHILE if COND4
      ......
      # COND4 transfere a execução para o fim do bloco while
  }
  # COND2 termina o bloco while
  ......
  # COND3 transfere a execução para o fim do bloco for
}
# COND3 termina ambos os blocos e o programa continua aqui...
```

Neste código fictício, examinam-se os elementos do array `@myarr` e executa-se um ciclo `while` quando um elemento contiver a letra `'x'`. Dentro deste bloco, várias condições transferem o processamento para o fim ou para fora de um ou dos dois blocos (veja os comentários no código). As `LABELS` também são utilizadas com a instrução `goto`.

Uma construção iterativa muito utilizada em *Perl*, que não existe em C, é `foreach $var LISTA`. Esta instrução itera nos elementos da lista atribuindo à variável escalar `$var` um elemento da lista de cada vez, e executando depois o bloco associado ao `foreach`. **Note**

que `$var` **varre** os **elementos** do array e não os índices! Se `$var` fôr omitida, é usada `$_` na iteração. Num script típico, é mais comum `foreach` do que o `for`. Vejamos um exemplo.

```
foreach $key (sort keys %myhash) {
  # processa os elementos do hash %myhash por ordem alfabética
  &processa($myhash{$key});
}
```

Se se fizer uma atribuição à variável de iteração do `foreach`, como por exemplo em `foreach $j (@lista) { ... $j = 'algo'; ...}`, vai-se modificar a própria lista pois `$j` é uma referência para cada elemento da lista.

As instruções de iteração são muito utilizadas nas linguagens de progamação, e o *Perl* não é excepção. Já vimos o `while` a ler ficheiros linha a linha, mas esta operação também pode ser feita caracter a caracter. O exemplo seguinte utiliza `until`:

```
open MYFILE, "bytes.txt";
$buffer = "";
until (eof(MYFILE)) {
  $buffer = join '', $buffer, getc(MYFILE);
  # é mais eficiente que $buffer .= getc(MYFILE) ;
  print "$buffer\n" if !eof(MYFILE);
}
```

## 5.3 Modificadores

Pode efectuar-se processamento condicional e iteração utilizando `if`, `unless`, `while` e `until` como modificadores. Às vezes esta forma é mais elegante e legível. A forma geral dos modificadores é:

```
if EXPR
unless EXPR
while EXPR
until EXPR
```

Por exemplo, é bastante legível o seguinte código (escrito em Inglês para concordar com os modificadores...):

```
&kiss_me() if $you_like_me;
$my_age = 30 unless &you_know_me();
$age++ while $age < 50;
$string .= "x" until length $string >= 20;
```

Se se pretender processar com modificadores um bloco de várias instruções, recorre-se à estrutura `do` {`BLOCO`}:

```
do {
  instr1;
  instr2;
  .......
} [if|unless|while|until] COND;
```

onde, na última linha, é seleccionado um modificador de entre os 4 possíveis.

## 5.4 Goto

O *Perl* implementa o 'famigerado' `goto` em três variedades: `goto LABEL`, `goto EXPR`, e `goto &NAME` (esta última não vai ser discutida).

A instrução `goto LABEL` tranfere incondicionalmente a execução do script para a instrução etiquetada `LABEL: instr`. Na segunda variedade, a expressão `EXPR` devolve o nome de uma `LABEL`, tratando-se de uma generalização da anterior. Esta forma permite implementar 'gotos' calculados (como no Fortran) do género de:

```
goto ("EMBORA", "CASA", "JANTAR")[$i];
```

em que, consoante o valor de `$i`, se segue para um dos destinos (i.e, 'labels') na lista.

# 6 Subprogramas, subrotinas ou funções

No *Perl* não há distinção entre subrotinas (ou rotinas), subprogramas e funções. É costume denominar-se estas entidades, indiferenciadamente, por qualquer daqueles nomes.

Pode criar-se uma subrotina (no exemplo, denominada `subnome`) em *Perl* de duas maneiras:

```
sub subnome { bloco }
sub subnome (proto) { bloco }
```

Na segunda forma, utiliza-se um protótipo `(proto)` para restringir o tipo e o número de argumentos fornecidos à subrotina. Se isso não fôr feito, o *Perl* aceita qualquer coisa! `bloco` é um bloco de código que constitui o corpo da subrotina. Os protótipos só são verificados se a função fôr invocada de determinada maneira (se quiser saber os detalhes consulte o manual `perlsub`).

Podem ser criadas **subrotinas anónimas** usando referências

```
$refsub = sub { bloco } ;    # esta subrotina não tem nome
```

e invocá-las como

```
%$refsub(args);
```

onde `args` são os eventuais argumentos.

Para criar uma função factorial (denominada `fact.pl`) basta criar o seguinte script:

```
# invocado como 'perl fact.pl num' calcula num!

print factorial(shift);

sub factorial {
  $_[0] < 2 ? 1 : $_[0] * factorial($_[0]-1);
}
```

O nível superior de execução deste script, muitas vezes denominado `'main'` por influência do C, contém uma única instrução: `print`. Disse-se, mais atrás, que as funções são invocadas com um nome começado por `&` mas isso não é exactamante verdade: se a função invocada tiver argumentos (como é o caso de `factorial()`) o *Perl* dispensa o `'&'` na invocação. Se a função não tiver argumentos, é obrigatório utilizar o prefixo `&`.

O que faz o `shift` no script? O `shift` tem um comportamento mágico quando é invocado sem argumento: no nível 'main' do script (i.e., fora de qualquer subrotina) aplica-se por defeito ao vector `@ARGV`; dentro duma subrotina, aplica-se por defeito ao vector `@_` (que tem um papel especial no interior de subotinas, como se verá). Conclui-se que a linha `print...` é equivalente a

```
print factorial(shift @ARGV);
```

ou, ainda a:

```
print factorial($ARGV[0]);
```

Vamos discutir agora o subprograma `factorial`. Um subprograma/função em *Perl* pode ser definido/declarado em qualquer parte do script. Poder-se-ia ter escrito primeiro a função factorial e invocá-la depois.

Não é visível a declaração de argumentos na função pois, implicitamente, os argumentos são passados para dentro da função através do array `@_` pela mesma ordem em que

aparecem na invocação da função. No presente caso, temos apenas um argumento escalar, que é passado para a primeira posição de `@_`, o escalar `$_[0]`. Este array `@_` tem uma 'cópia local' em cada subprograma: pode-se invocar rotinas dentro de rotinas — aliás, neste caso `factorial()` é uma função **recursiva** invocada $n$ vezes —, pois criam-se 'cópias' locais de `@_` em cada invocação.

A única instrução na função é o operador ternário de selecção condicional, cujo protótipo é `cond ? instr_yes : instr_no;`. Se a condição `cond` fôr verdadeira, é executada a instrução `instr_yes` e, se não fôr, é executada `instr_no`. Já se verá a utilização do operador `if` como alternativa a este operador.

A função `factorial` que definimos é 'mínima' pois não faz qualquer verificação de erros e aceita alegremente argumentos negativos, não inteiros, ou não existentes. Veja a mesma função escrita de uma forma mais clara:

```
sub factorial {
  my($n) = shift @_;  # ou 'my($n) = $_[0];'
  if ($n < 2) {
      return 1;
  } else {
      return $n * &factorial($n-1);
  }
}

print &factorial($ARGV[0]);
```

Nesta versão 'para iniciados' `8-)` tudo é claro! Temos um `if` e temos a instrução `return` que nos faz chorar de saudade do C! Na anterior versão para 'profissionais' não existia instrução `return`: em *Perl*, um subprograma devolve o último valor/objecto que calculou (pode devolver um array ou um hash, por exemplo).

Uma função que devolve um array terminará da seguinte forma:

```
sub devolve_array {
  ............
  @array_que_sai;    # o mesmo que 'return @array_que_sai'
}
```

Obviamente, este subprograma poderia devolver um escalar `$escalar` ou um hash `%hash`, terminando a execução com a simples invocação do objecto. Um subprograma pode ter múltiplas saídas como acontece com a função `factorial`.

## 6.1 Variáveis locais

Quando se mencionam variáveis 'novas' num subprograma, elas são criadas como variáveis **globais**. Isto significa que vão ficar para sempre a ocupar memória de sistema, mesmo que o subprograma seja executado uma única vez. Por vezes, pretende evitar-se este comportamento, não só para economizar memória como também para evitar 'colisões' entre variáveis utilizadas em blocos de código separados por milhares de linhas, o que acontece em 'projectos da vida real' `8-)`.

Para declarar variáveis localmente, dentro de subprogramas ou de blocos, o *Perl* dispõe de duas instruções de declaração: `my` e `local`.

`my` declara as variáveis como privadas dentro do bloco (o jargão técnico é 'lexically scoped'). Elas são 'suprimidas' quando termina a execução do bloco/subrotina e são invisíveis em qualquer outro ponto do programa.

Por outro lado, `local` é usado para declarar variáveis 'dinâmicas'. Estas variáveis não são privadas do bloco, mas sim variáveis globais com 'valores locais'. Quando a subrotina é executada o valor global é escondido e utilizado o valor local. As variáveis declaradas com `local` são visíveis em subrotinas invocadas de dentro do bloco/subrotina onde foram declaradas. Veja-se o seguinte exemplo, onde se ilustra a diferença entre `my` e `local`..

```
$a = "a_global";    # variável global
&printa;            # 1ª invocação
&submy;
&subloc;

sub submy {
  my $a = "a_my";
  &printa;        # 2ª invocação
}

sub subloc {
  local $a = "a_local";
  &printa;        # 3ª invocação
}

sub printa {
  print $a , "\n";     # imprime a variável $a visível em printa
}
```

A execução deste programa dá o seguinte resultado:

```
a_global
a_global
a_local
```

que confirma aquilo que se disse. A 1ª invocação imprime obviamente a variável global. A 2ª invocação é feita de dentro de `submy` onde se declarou localmente `$a` com `my` e que, portanto, só existe dentro desta subrotina `submy`: assim, o subprograma `printa` 'vê' a variável `$a` global e imprime `a_global`. Finalmente, a 3ª invocação é feita dentro de `subloc` que declarou localmente `$a` com `local` e, assim, torna visível este `$a` em todas as subrotinas invocadas no corpo da subrotina `subloc`: consequentemente, `printa` vê a variável `$a` declarada com `local` e imprime `a_local`.

Na prática, o que normalmente se pretende com a utilização de variáveis locais é o comportamento associado à declaração `my`, pelo que esta é a mais utilizada.

## 6.2 Passagem de argumentos para subrotinas

Para melhorar a legibilidade do código, costuma fazer-se uma 'cópia' para variáveis locais do array `@_` que contém os argumentos passados para a função. Quando se pretende

passar para dentro do subprograma um ou mais arrays juntamente com outras variáveis, é preciso ter algum cuidado. Suponha que se criaram as seguintes variáveis

```
$e1 = 10.0;
$e2 = 'boo';
@a1 = (1,2,3,4);
@a2 = ('a','b','c');
%h  = ('alfa' => 'romeo', 'land' => 'rover');
```

e que se pretende enviá-las como argumentos para uma função denominada `subprog`. Veja o seguinte bloco de possíveis invocações (numeradas 'artificialmente') e os respectivos comentários (onde o símbolo `'<-'` significa 'recebe' ou 'fica igual a').

```
# ((1))
&subprog($e1,$e2);
.......................
sub subprog {
  my ($a,$b) = @_;     # Correcto: $a <- $e1 e $b <- $e2
.......................

# ((2))
&subprog($e1,@a1);
.......................
sub subprog {
  my ($a,@b) = @_;     # Correcto: $a <- $e1 e @b <- @a1
.......................

# ((3))
&subprog($e1,%h);
.......................
sub subprog {
  my ($a,%b) = @_;     # Correcto: $a <- $e1 e %b <- %h
.......................

# ((4))
&subprog(@a1,$e1);
.......................
sub subprog {
  my (@a,$b) = @_;     # Incorrecto: @a <- (@a1,$b) e $b <- ''
.......................

# ((5))
&subprog(@a1,@a2);
.......................
sub subprog {
  my (@a,@b) = @_;     # Incorrecto: @a <- (@a1,@a2) e @b <- ()
.......................

# ((6))
&subprog(%h,@a1);
.......................
sub subprog {
  my (%a,@b) = @_;     # Incorrecto: %a <- (%h,@a1) e @b <- ()
```

```
.......................
```

Como é visível nos exemplos ((4)), ((5)) e ((6)), quando a primeira variável a recolher argumentos no `my()` é uma lista, irá 'engolir' todo o vector `@_`. Para resolver este problema, usam-se referências (que são escalares), pois com argumentos escalares não há problema. Veja os exemplos ((4)) e ((5)) corrigidos com referências:

```
# ((4'))
&subprog(\@a1,$e1);
.......................
sub subprog {
    my ($a,$b) = @_;     # Correcto: $a <- \@a1 e $b <- $e1
    @a = @$a;            # De-refere-se $a=\@a1
.......................

# ((5'))
&subprog(\@a1,\@a2);
.......................
sub subprog {
    my ($a,$b) = @_;     # Correcto: $a <- \@a1 e $b <- \@a2
    @a = @$a;            # De-refere-se
    @b = @{$b};          # De-refere-se com chavetas para clarificar...
.......................
```

Bem, já foi revista a matéria mais importante relacionada com subprogramas. Vamos agora discutir a execução de código 'on-the-fly' (i.e., imediatamente), código esse eventualmente criado durante a execução do próprio script.

# 7 Execução de código em 'run-time': `eval` e `require`

Uma das vantagens das linguagens interpretadas é que se pode escrever código durante a execução de um script e utilizá-lo imediatamente. O *Perl* possui a função `eval` para executar código escrito numa string. O mesmo pode ser conseguido com a escrita de código num ficheiro externo que é executado com `require` ou `use`. Adicionalmente, `eval` é o mecanismo existente no *Perl* que permite lidar com excepções. Examine o seguinte script (denominado `m.pl`.)

```
$a = 10 ;
$b = 0 ;
$c = $a / $b ;
print $a, $b, $c ;
```

Quando executado, este script termina prematuramente com o erro
`Illegal division by zero at m.pl line 3.`
pois uma divisão por zero é uma excepção suficientemente grave para abortar o programa.

Como é que se pode testar, num bloco de código, a divisão por zero ou, em termos gerais, qualquer outra operação ilegal, sem abortar o script? Usando `eval`, pois claro!

Dentro do `eval` o código é executado numa 'redoma': se não houver qualquer erro, após a sua execução a variável `$@` é igual à string nula ; se houver erro, esta variável contém uma string com a mensagem de erro. Este processo permite lidar com excepções. Observe como poderia 'proteger' o anterior bloco de código com `eval`:

```
$a = 10 ;
$b = 0 ;
eval {$c = $a / $b};
# ou eval '$c = $a / $b';
$c="infty" if $@;
print "a=$a  b=$b  c=$c\n";
```

Quando executado, este script escreve na consola `a=10  b=0  c=infty`.

A divisão por zero é executada dentro do `eval`; como há erro, é escrita a respectiva mensagem de erro em `$@` que, quando é testada após o `eval`, é a uma condição verdadeira (se não tivesse havido qualquer erro, `$@=''` era falsa). Então, executa-se a atribuição `$c="infty"`, o valor imprimido no final. Note que a execução do script não foi interrompida pelo erro dentro do `eval`.

A função `eval` pode ser utilizada de duas maneiras:
**eval** `BLOCO`
**eval** `EXPR`
A principal diferença é que o uso do `BLOCO` (código entre chavetas) é mais eficiente do que o uso da expressão `EXPR`, pois o bloco só é compilado uma vez: esta forma deve ser sempre utilizada se o código embebido na string não variar. No anterior script qualquer das formas poderia ter sido utilizada (a expressão `eval '$c = $a / $b';` está comentada).
Pode aplicar `eval` a uma variável escalar de duas formas equivalentes:

```
$s = '$c = $a / $b'
eval $s;
```

```
eval "$s";      # forma equivalente
```

Veja agora um exemplo que mostra como escrever código num ficheiro externo e proceder à sua execução imediata no mesmo script.

```
open(CUBO,">cubo.pl") || die 'não posso abrir cubo.pl!\n' ;
print CUBO 'sub cubo { $_[0] ** 3 } 1' ;    # escreve um subprograma no ficheiro
close CUBO ;
require 'cubo.pl' ;        # executa cubo.pl
# eval `type cubo.pl` ;    # em Windows
# eval `cat cubo.pl` ;     # em Unix/Linux
# eval qx/type cubo.pl/ ;  # em Windows
# eval qx/cat cubo.pl/ ;   # em Unix/Linux
# NOTA: quando se usa eval não é necessário terminar cubo.pl com '1'
print cubo(10), "\n" ;     # imprime 1000
```

Abre-se o ficheiro `cubo.pl` e este é associado ao filehandle `CUBO` onde é escrita a subrotina `cubo`. Repare que é escrito `1` no fim deste ficheiro, para que o valor 'lógico' devolvido por `require` seja verdadeiro: qualquer número ou string diferentes de `0` ou de `""` (a string nula) poderia ter sido usado em vez do `1`.

Após o ficheiro ser processado com `require`, todas as suas variáveis e subprogramas ficam visíveis no actual contexto, como se tivessem sido escritos no actual script.

Pode-se realizar a mesma tarefa com `eval`, utilizando a forma especial de string com acentos graves `` `comando` `` (ou `qx/comando/`) que executa o comando de sistema e devolve o resultado. O resultado da execução de um `type` ou de um `cat` (em Unix/Linux) é a listagem do ficheiro, que é então processado pelo `eval`. Isto é equivalente a uma invocação com `require`. Se se utilizar `eval`, fica-se dispensado de colocar um valor 'verdadeiro' no fim do ficheiro (1, no caso de `cubo.pl`).

Finalmente, há formas bem mais sofisticadas de utilizar código escrito em ficheiros externos. Veja as instruções/declarações `use`, `import`, `package` e `do` . Estude também 'programação orientada para objectos' em *Perl*.

Vamos passar agora para 'algo completamente diferente' (isto é uma piada aos Monty Python): as expressões regulares.

# 8 Expressões regulares (*regexes*)

Um dos pontos fortes do *Perl*, que justifica parcialmente a sua popularidade, é a sua excelência no processamento de texto. Essa excelência está intimamente ligada ao suporte de expressões regulares embebido na linguagem e à qualidade do respectivo 'motor' ('regex engine'). É frequente referir as expressões regulares apenas por **regex** ou **regexp** (plural **regexes** ou **regexps**).

Nas expressões regulares podem utilizar-se caracteres e metacaracteres. Os caracteres ordinários são intrepretados como eles próprios. Os metacaracteres têm um significado especial. Em particular, os seguintes metacaracteres têm o seguinte comportamento (semelhante àquele encontrado nas expressões regulares da ferramenta `egrep` em sistemas Unix).

| **Metacaracter** | **Significado** |
|---|---|
| `\` | Escapa (ou cita) o próximo caracter ou metacaracter |
| `^` | Acerta no princípio da linha |
| `.` | Acerta em qualquer caracter (excepto em newline `\n`) |
| `$` | Acerta no fim da linha (ou antes de `'\n'` no fim) |
| `\|` | Alternância |
| `()` | Agrupamento |
| `[]` | Classe de caracteres |

Por defeito, o caracter `"^"` só acerta no início da string e o caracter `"$"` no fim (ou antes do `'\n'` no fim). Caracteres `'\n'` não são atingidos por `"^"` ou por `"$"`. Pode-se, no entanto, tratar a string como se tivesse várias linhas utilizando o modificador `/m` no fim da regex. Veja, mais abaixo, a definição destes modificadores.

Para simplificar as substituições em mais que uma linha, o caracter `"."` nunca acerta em `'\n'` a menos que seja usado o modificador `/s` que diz ao *Perl* que a string consiste de uma única linha (mesmo que isso não seja verdade).

São reconhecidos os seguintes quantificadores em regexes:

| **Quantificador** | **Significado** |
|---|---|
| `*` | Acerta 0 ou mais vezes |
| `+` | Acerta 1 ou mais vezes |
| `?` | Acerta 1 ou 0 vezes |
| `{n}` | Acerta exactamente `n` vezes |
| `{n,}` | Acerta pelo menos `n` vezes |
| `{n,m}` | Acerta pelo menos `n` mas não mais do que `m` vezes |

(Se uma chaveta aparece noutro contexto, é tratada como um caracter normal.) O quantificador `"*"` é equivalente a `{0,}`, o quantificador `"+"` é equivalente a `{1,}` e o quantificador `"?"` é equivalente a `{0,1}`.

Por defeito, um sub-padrão quantificado numa regex é ganancioso, pois tenta acertar ((ou coincidir) no maior número possível de caracteres (numa dada posição) e simultaneamente permitir ao restante padrão na regex coincidir. Se se pretender que o sub-padrão acerte um número mínimo de vezes, acrescenta-se ao quantificador `'?'`. O significado dos quantificadores não muda com o `'?'`, apenas a sua "ganância".

| Quantificador | Significado |
|---|---|
| `*?` | Acerta 0 ou mais vezes |
| `+?` | Acerta 1 ou mais vezes |
| `??` | Acerta 0 ou 1 vez |
| `{n}?` | Acerta exactamente `n` vezes |
| `{n,}?` | Acerta pelo menos `n` vezes |
| `{n,m}?` | Acerta pelo menos `n` mas não mais que `m` vezes |

Uma vez que os padrões são processados como strings interpoláveis, os seguintes caracteres especiais são reconhecidos em regexes:

| Caracter | Significado | Representação |
|---|---|---|
| `\t` | tab | (HT, TAB) |
| `\n` | newline | (LF, NL) |
| `\r` | return | (CR) |
| `\f` | form feed | (FF) |
| `\a` | alarme | (BEL) |
| `\e` | escape | (ESC) |
| `\033` | caracter em octal | |
| `\x1B` | caracter em hexadecimal | |
| `\x{263a}` | caracter hexadecimal generalizado | (SMILEY em Unicode) |
| `\c[` | caracter control | |
| `\l` | passa próximo caracter a minúscula | |
| `\u` | passa próximo caracter a maiúscula | |
| `\L` | minúsculas até `\E` | |
| `\U` | maiúsculas até `\E` | |
| `\E` | fim de modificação `\L, \U, \q` | |
| `\Q` | inibe metacaracteres até `\E` | |

Adicionalmente, em *Perl* definem-se as seguintes classes de caracteres (há outras, relacionadas com a codificação Unicode):

| Classe | Significado |
|---|---|
| `\w` | Acerta num caracter de 'palavra' (alfanuméricos mais `"_"`) |
| `\W` | Acerta num caracter que não seja de 'palavra' |
| `\s` | Acerta num caracter espaço (tab,newline,espaço,...) |
| `\S` | Acerta num caracter que não seja espaço |
| `\d` | Acerta num dígito |
| `\D` | Acerta num caracter que não seja dígito |

`\w` coincide com um único caracter alfanumérico, não uma palavra. Use `\w` para acertar numa string constituída por identificadores de *Perl* (diferente de uma palavra em Português, devido aos caracteres acentuados).

`\w, \W, \s, \S, \d` e `\D` podem ser utilizadas dentro de classes de caracteres mas não podem definir o princípio ou o fim de gamas. Veja o operador de transliteração `tr///`.

As seguintes 'afirmações' ('assertions') de comprimento zero (i.e., não acertam num caracter, mas sim na fronteira da string) estão definidas em *Perl*:

| Classe | Significado |
|---|---|
| \b | Acerta numa fronteira de palavra |
| \B | Acerta fora de uma fronteira de palavra |
| \A | Acerta somente no princípio da string |
| \Z | Acerta somente no fim da string, ou antes de '\n' no fim |
| \z | Acerta somente no fim da string |

A fronteira de uma palavra é o ponto (imaginário...) que tem um `\w` de um lado e um `\W` do outro lado. O início e o fim da string são considerados como `\W`.

`\A` e `\Z` são como `"^"` e `"$"`, excepto que não acertam múltiplas vezes quando o modificador `/m` é usado, enquanto que `"^"` e `"$"` acertam em 'fronteiras' de linha internas. Para acertar o verdadeiro fim da string e não ignorar o `'\n'` que lá possa estar use `\z`.

A construção com parêntesis curvos `(...)` cria variáveis de captura de substrings. Para se referir à `<dígito>`-ésima destas variáveis (por ordem de aparição na regex) usa-se `\<dígito>` dentro da mesma regex ou `$<dígito>` fora da regex. A referência a uma anterior coincidência chama-se *retro-referência* ('backreference'). O limite de retro-referências é muito grande. (Nota: o *Perl* desambigua inteligentemente retro-referências superiores a 9, e.g. a retro-referência `\12` e o mesmo valor como sósia de `012` em octal, que se refere ao 10º caracter ASCII. Consulte o manual).

Exemplos:

```
s/^([^ ]*) *([^ ]*)/$2 $1/;     # troca as duas primeiras palavras
if (/(.)\1/) {                  # encontra o primeiro caracter duplo
     print "'$1' é o primeiro caracter duplo\n";
}

if (/Tempo: (..):(..):(..)/) {   # extrai valores
    $horas = $1;
    $minutos = $2;
    $segundos = $3;
}
```

Há variáveis especiais relacionadas com regexes. `$+` devolve o último acerto com parêntesis. `$&` devolve toda a string atingida. `$‘` devolve os caracteres anteriores à string atingida. E `$'` devolve todos os caracteres posteriores à string atingida. O uso destas variáveis especiais e de `$1, $2, ...` **atrasa MUITO** o processamento das expressões regulares no script: use-as somente se delas necessitar.

Em *Perl* os caracteres escapados com `'\'` são sempre alfanuméricos. Assim, contruções como `\\, \(, \), \<, \>, \{,` ou `\}` são sempre interpretadas literalmente i.e., não têm qualquer significado especial pois não são metacaracteres.

Usa-se a função `quotemeta()` ou os escapes `\Q...\E` para inibir localmente os significados especiais dos metacaracteres, da seguinte forma:

```
/$unquoted\Q$quoted\E$unquoted/
```

## 8.1 Sintaxe ampliada ('extended') de expressões regulares

O *Perl* suporta uma sintaxe ampliada para expressões regulares que consiste de um par de parêntesis com um ponto de interrogação como primeiro caracter `(?...)`. O caracter que se segue ao ponto de interrogação indica o tipo da extensão. Consulte a secção 'Extended Patterns' no manual `perlre`.

## 8.2 Regras aplicáveis em expressões regulares

Eis um sumário das regras que se aplicam em expressões regulares.

Qualquer caracter simples acerta nele próprio, a menos que seja um metacaracter com significado especial. Pode forçar-se um metacaracter a ser interpretado literalmente prefixando-o com `"\"` (e.g., `"\."` acerta apenas num simples ponto, `"\\"` acerta em `"\"`, etc). Uma sequência de caracteres acerta apenas nessa mesma sequência de caracteres.

Pode especificar-se uma classe de caracteres encerrando a respectiva lista entre `[ ]`, que assim acertará em qualquer caracter da lista. Se o primeiro caracter após `[` fôr `"^"` a classe acerta em cada caracter que **não pertença** à lista (a classe é negada). Numa lista, o caracter `"-"` especifica uma gama de forma que `a-z` representa todos as letras minúsculas. Se se pretender um `"-"` ou um `"["` incluído na classe, deve-se pô-lo no início da lista (ou após um `"^"`) ou escapá-lo com `"\"`. `"-"` também é interpretado literalmente quando no fim duma lista (antes do `"]"`). Em princípio, as gamas de caracteres só deverão ser aplicadas a caracteres alfanuméricos.

Certos caracteres podem ser especificados por metacaracteres: `'\n'` é um 'newline', `'\t'` é um 'tab', etc. Mais geralmente `\nnn`, onde `nnn` é uma string de dígitos octais, representa o caracter ASCII de valor `nnn`. De forma semelhante `\xnn`, onde `nn` é uma string de dígitos hexadecimais, representa o caracter ASCII de valor `nn`. A expressão `\cx` representa o caracter ASCII `control-x`. Finalmente, o metacaracter `"."` acerta em qualquer caracter excepto em `'\n'` (a menos que se use o modificador da regex `/s`).

Pode especificar-se alternativas num padrão duma regex com `"|"` a separá-las: assim, `fee|fie|foe` coincide em `"fee"`, em `"fie"` ou em `"foe"` na string alvo (assim como a regex equivalente `"f(e|i|o)e"`). Os delimitadores exteriores das alternativas são parêntesis curvos (e.g. `(fee|fie|foe)`) e é comum adicionar os parêntesis para definir com clareza os limites.

As alternativas são testadas da esquerda para a direita; a primeira a ser encontrada que conduz a uma coincidência da regex é a escolhida.

Dentro de uma classe `[...]` `"|"` é interpretada literalmente, por forma que `[fee|fie|foe]` é equivalente a `[feio|]` (os caracteres repetidos dentro de uma classe não interessam).

## 8.3 Pesquisa, substituição e transliteração

As expressões regulares podem ser utilizadas em três funções importantes: para pesquisar texto, para substituir texto e para transliterar texto (substituir grupos de caracteres directamente por outros). Na exposição que se segue, `PATTERNx` (onde `x` é um dígito ou não existe) é um padrão de expressão regular.

- **m/PATTERN/cgimosx**
  **/PATTERN/cgimosx**
  Nesta construção pesquisa-se numa string a regex `PATTERN`. Em contexto escalar, a construção dá verdadeira se a pesquisa fôr bem sucedida e falsa em caso contrário. Se não fôr especificada uma string com os operadores `=~` ou `!~`, a string `$_` é examinada. A string à esquerda destes operadores pode ser o resultado de uma expressão. Na forma `m/.../`, as barras podem ser substituídas por outros 2 caracteres não alfanuméricos, iguais entre si, ou por um par de parêntesis complementares. Exemplos:

```
if (m[Pedro]i) ...       # verdade se $_ contém pedro ou Pedro ou Pedroso...
if ($_ =~ m|Pedro|i) ... # equivalente
if ($linha !~ /tua/) ... # verdade se $linha não contém tua
                         # O m em m/.../ foi omitido
```

As opções, ou modificadores, têm o seguinte significado:

| Modificador | Significado |
|---|---|
| c | Não faz reset à posição de pesquisa num insucesso com /g activo |
| g | Pesquisa globalmente, i.e., encontra todas as ocorrências |
| i | Pesquisa insensível a maiúscula/minúscula |
| m | Trata a string como linhas múltiplas |
| o | Compila PATTERN uma única vez |
| s | Trata a string como uma única linha |
| x | Usa expressões regulares generalizadas ou ampliadas ('extended') |

Se o delimitador da regex fôr `"/"`, o `m` inicial pode ser omitido. Especificando o `m`, pode utilizar-se qualquer par de caracteres não alfanuméricos. Se o delimitador fôr `"?"` , aplica-se a regra 'acerta-apenas-uma-vez' (na forma `?PATTERN?)`. Se o delimitador fôr uma plica `"'"` não é feita interpolação.

`PATTERN` pode conter variáveis, que serão interpoladas (e a regex recompilada) excepto quando o delimitador é a plica `"'"`. Utilizando o modificador `/o` a compilação é feita apenas uma vez (poupa-se tempo!): pode ser aplicado quando a regex é constante. Aqui presume-se que as variáveis interpoladas não mudam (se mudaram, o *Perl* nem dá por isso!) Veja também `qr//`.

Se a opção `/g` não fôr usada, em contexto de lista `m//` devolve uma lista que consiste das subexpressões encontradas com os parêntesis, i.e. `($1, $2, $3...)`. Se não há parêntesis no padrão, o valôr devolvido é a lista caso a pesquisa seja bem sucedida. Com ou sem parêntesis, após um insucesso é devolvida uma lista vazia.

Exemplos:

```
open(TTY, '/dev/tty');
<TTY> =~ /^y/i && foo();    # executa foo() se a 1ª condição fôr verdadeira

if (/Version: *([0-9.]*)/) { $version = $1; }       # extrai versão

next if m#^/usr/spool/uucp#;

# grep do pobrezinho...
$arg = shift;
```

```
while (<>) {
    print if /$arg/o;       # compila apenas uma vez
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

Este último exemplo parte `$foo` nas primeiras duas palavras e no resto da linha, e atribui estas 3 peças a `$F1`, `$F2` e `$Etc`. A condição é verdadeira se qualquer uma das 3 variáveis fôr atribuída.

O modificador `/g` especifica globalidade, i.e., acerta tantas vezes quanto as possíveis na string. Em contexto escalar, pode recuperar-se a posição após o último acerto com a função `pos()`. Consulte o manual.

- **s/PATTERN/REPLACEMENT/egimosx**

Pesquisa o padrão `PATTERN` numa string e, se o encontra, substitui-o com `REPLACEMENT` (substituto). Devolve o número de substituições feitas. Se não substituíu nada, devolve a string nula (i.e., uma condição falsa).

Se não se especificar uma string com os operadores `=~` ou `!~`, é pesquisada e modificada a variável `$_`. Se especificada, a string deve ser um escalar, elemento de array ou de hash ou uma atribuição a um deles.

Se o delimitador fôr uma plica, não é feita interpolação nem em `PATTERN` nem em `REPLACEMENT` (o modificador `/e` altera este comportamento). Com qualquer outro delimitador, se existe um `$...` que se assemelha a uma variável, esta será interpolada durante a execução. O modificador `/o` compila a regex apenas uma vez, na primeira interpolação da variável.

Os modificadores são:

| Modificador | Significado |
|---|---|
| e | Calcula o lado direito como uma expressão |
| g | Substitui globalmente, i.e., substitui todas as ocorrências |
| i | Pesquisa insensível a maiúscula/minúscula |
| m | Trata a string como linhas múltiplas |
| o | Compila PATTERN uma única vez |
| s | Trata a string como uma única linha |
| x | Usa expressões regulares generalizadas ou ampliadas ('extended') |

Qualquer caracter não-alfanumérico e não espaço, pode substituir os delimitadores `"/"`. Se `PATTERN` fôr delimitada por parêntesis de qualquer tipo, `REPLACEMENT` também ganha um par de delimitadores (de qualquer tipo), e.g., `s(foo)(bar)` ou `s<foo>/bar/`. O modificador `/e` faz com que `REPLACEMENT` seja tratada como uma expressão *Perl* que é executada. A sua sintaxe é verificada logo na compilação.

Exemplos:

```
s/\bbom\b/mau/g;                          # não substitui bombom...
$path =~ s|/usr/bin|/usr/local/bin|;
s/Login: $foo/Login: $bar/;               # regexes construídas na execução
($foo = $bar) =~ s/this/that/;            # copia $bar primeiro, depois muda $foo
$count = ($paragraph =~ s/Mister\b/Mr./g); # conta o número de substituições
```

```
$_ = 'abc123xyz';
s/\d+/$&*2/e;               # dá 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e;  # dá 'abc  246xyz'
s/\w/$& x 2/eg;             # dá 'aabbcc  224466xxyyzz'

s/%(.)/$percent{$1}/g;              # muda escapes 'por-cento'; sem /e
s/%(.)/$percent{$1} || $&/ge;       # substitui em cada execução, com /e
s/^=(\w+)/&pod($1)/ge;              # usa chamada a função

# expande variáveis dinâmicas em $_ com referência simbólica
s/\$(\w+)/${$1}/g;

# Adiciona 1 ao valor de todos os números na string
s/(\d+)/1 + $1/eg;

# Apaga comentários num programa em C.
$program =~ s {
    /\*     # Acerta no delimitador de abertura.
    .*?     # Acerta num número mínimo de caracteres.
    \*/     # Acerta no delimitador de fecho.
} []gsx;

s/^\s*(.*?)\s*$/$1/;   # retira espaços no início e no fim de $_, caro
for ($variable) {      # retira espaços iniciais e finais a $variable, barato
    s/^\s+//;
    s/\s+$//;
}

s/([^ ]*) *([^ ]*)/$2 $1/;  # troca os primeiros dois campos
```

Note o uso de `'$'` em vez de `'\'` no último exemplo. A forma `\<digit>` só é usada no lado esquerdo do operador de substituição: em qualquer outro ponto usa-se `$<digit>`.

- **tr/SEARCHLIST/REPLACEMENTLIST/cdsUC**
  **y/SEARCHLIST/REPLACEMENTLIST/cdsUC**
  Translitera todas as ocorrências dos caracteres encontrados na lista `SEARCHLIST` com o caracter na posição correspondente da lista de substituição `REPLACEMENTLIST`. Devolve o número de caracteres substituídos ou apagados. Se não fôr especificada uma string com os operadores `=~` ou `!~`, é transliterada `$_`. A string a transliterar deverá ser um escalar, um elemento de array ou de hash, ou uma atribuição a um deles.

  Uma gama de caracteres é especificada com `"-"`, de forma que `tr/A-J/0-9/` é o mesmo que `tr/ACEGIBDFHJ/0246813579/`. `y` é sinónimo de `tr`. Em princípio, dever-se-á usar gamas apenas com caracteres alfanuméricos e não com outros caracteres (e.g., de pontuação).

  Se `SEARCHLIST` fôr delimitada por qualquer tipo de parêntesis, então `REPLACEMENTLIST` tem o seu próprio par de delimitadores, que podem ou não ser parêntesis, e.g., `tr[A-Z][a-z]` ou `tr(+\-*/)/ABCD/`.

  O significado dos modificadores de `tr///` é:

| Modificador | Significado |
|---|---|
| c | Complementa a SEARCHLIST. |
| d | Apaga caracteres encontrados mas não substituídos. |
| s | Comprime caracteres substituídos em duplicado. |
| U | Translitera para/de UTF-8. |
| C | Translitera para/de caracter de 8 bits (octeto). |

Se o modificador `/c` fôr especificado, o conjunto de caracteres em `SEARCHLIST` é complementado. Se o modificador `/d` fôr especificado, quaisquer caracteres em `SEARCHLIST` não encontrados em `REPLACEMENTLIST` são apagados. Se o modificador `/s` fôr especificado, sequências de caracteres transliteradas para o mesmo caracter são comprimidas para uma única instância do caracter.

Se o modificador `/d` fôr usado, a `REPLACEMENTLIST` é sempre interpretada exactamente como especificado. Caso contrário, se a `REPLACEMENTLIST` fôr mais curta que a `SEARCHLIST`, o último caracter é replicado até igualar o tamanho das duas listas. Se a `REPLACEMENTLIST` estiver vazia, a `SEARCHLIST` é replicada. Esta última possibilidade é útil para contar caracteres duma classe ou para comprimir sequências de caracteres numa classe.

Exemplos:

```
$ARGV[1] =~ tr/A-Z/a-z/;    # passa $ARGV[1] para minúsculas
$cnt = tr/*/*/;             # conta os asteriscos em $_
$cnt = $ceu =~ tr/*/*/;     # conta as 'estrelas' no $ceu
$cnt = tr/0-9//;            # conta os dígitos em $_
tr/a-zA-Z//s;               # bookkeeper -> bokeper
($HOST = $host) =~ tr/a-z/A-Z/;
tr/a-zA-Z/ /cs;             # substitui não-alfanuméricos por espaço
tr [\200-\377]
   [\000-\177];             # apaga o oitavo bit
tr/\0-\xFF//CU;             # muda Latin-1 para Unicode
tr/\0-\x{FF}//UC;           # muda Unicode para Latin-1
```

Se houver múltiplas transliterações para um caracter, apenas a primeira é utilizada:

```
tr/AAA/XYZ/
```

translitera qualquer `A` para `X`.

Uma vez que a tabela de transliteração é construída durante a compilação, nem a `SEARCHLIST` nem a `REPLACEMENTLIST` são sujeitas a interpolação de variáveis. Isto significa que se se pretender utilizar variáveis deve-se usar `eval()`:

```
eval "tr/$oldlist/$newlist/";
die $@ if $@;

eval "tr/$oldlist/$newlist/, 1" or die $@;
```

# 9 Manipulação de arrays de arrays em Perl

## 9.1 Declaração e acesso a arrays de arrays (AoAs)

A estrutura de dados composta mais simples de construir é o array de arrays. O processo de criação é simples de compreender e as técnicas aplicadas são válidas para estruturas de dados mais avançadas.

Um array de arrays é um comum array `@AoA` que pode ser construído implicitamente utilizando dois índices, como `$AoA[3][2]`. Para construir e preencher explicitamente o array faz-se o seguinte:

```
# construir um array de referências para arrays
@AoA = (
       [ "fred", "barney" ],
       [ "george", "jane", "elroy" ],
       [ "homer", "marge", "bart" ],
);
print $AoA[2][2];                    # imprime bart
```

É necessário ter em atenção que os parêntesis externos são curvos, pois estamos a preencher um `@array`. Se se pretendesse aceder ao array, anonimamente, através de uma referência, far-se-ia o seguinte:

```
# construir uma referência para um array de referências para arrays
$ref_to_AoA = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];
print $ref_to_AoA->[2][2];            # imprime elroy
```

Repare que os parêntesis exteriores mudaram para parêntesis rectos, e a sintaxe de acesso também mudou. Em *Perl*, contrariamente ao C, não se pode utilizar indiscriminadamente, como sinónimos, arrays e referências para arrays. `$ref_to_AoA` é uma referência para um array enquanto que `@AoA` é mesmo um array. Assim, `$AoA[2]` não é um array mas uma referência para um array.

Então, porque é que se escreve

```
$AoA[2][2]
$ref_to_AoA->[2][2]
```

em vez de se escrever

```
$AoA[2]->[2]
$ref_to_AoA->[2]->[2]
```

Bem, porque existe a regra de que entre parêntesis adjacentes (parêntesis rectos ou chavetas) é-se livre de omitir a seta de dereferência do apontador. Mas a **primeira** (entre o nome do apontador e o primeiro parêntesis) **não pode ser omitida**, o que significa que `$ref_to_AoA` precisa sempre de a utilizar.

## 9.2 Criação dinâmica de AoAs

Vimos como se pode criar implicitamente uma estrutura de dados `@AoA` fixa. Vamos ver agora como criá-la e como se pode adicionar-lhe elementos dinamicamente.

Primeiro, vamos ver como se pode lê-la de um ficheiro, isto é, como adicionar uma linha de cada vez. Assume-se que há um ficheiro cujas linhas irão corresponder a uma linha do array, e que cada palavra da linha é um elemento. Se se pretende criar um `@AoA` com o conteúdo deste ficheiro, eis como fazê-lo correctamente:

```
while (<>) {
    @temp = split;
    push @AoA, [ @temp ];
}
```

Também se poderia preencher o array com o resultado de uma função:

```
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ umafunc($i) ];
}
```

Ou poder-se-ia utilizar uma variável temporária para clarificar o código:

```
for $i ( 1 .. 10 ) {
    @temp = umafunc($i);
    $AoA[$i] = [ @temp ];
}
```

É extremamente importante que se utilize o construtor de referências para arrays `[ ]` porque

```
$AoA[$i] = @temp;       # Errado!
```

está errado, pois a atribuição de um array em contexto escalar devolve o seu número de elementos, o que provavelmente não é o que se pretende.

Se estiver activado o pragma `use strict;` terão que ser acrescentadas algumas declarações:

```
use strict;
my(@AoA, @temp);
while (<>) {
    @temp = split;
    push @AoA, [ @temp ];
}
```

Claro que o array temporário não necessita de existir...

```
while (<>) {
    push @AoA, [ split ];
}
```

Se se souber qual a posição `$i` onde se quer colocar a referência, usa-se:

```
my (@AoA, $i, $line);
for $i ( 0 .. 10 ) {
    $line = <>;
    $AoA[$i] = [ split ' ', $line ];
}
```

ou mesmo, apenas

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split ' ', <> ];
}
```

Porém, não se deve abusar do uso de funções que potencialmente podem devolver listas, em contexto escalar, sem o indicar explicitamente. O seguinte código seria mais claro:

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split ' ', scalar(<>) ];
}
```

Se se pretendesse utilizar uma referência `$ref_to_AoA`, ter-se-ia que escrever algo como:

```
while (<>) {
    push @$ref_to_AoA, [ split ];
}
```

Já viu como pode adicionar linhas ao array de arrays. Mas como se adicionam colunas? Se se está na presença de matrizes, é simples utilizar a atribuição directa:

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        $AoA[$x][$y] = func($x, $y);
    }
}

for $x ( 3, 7, 9 ) {
    $AoA[$x][20] += func2($x);
}
```

Não interessa se os elementos já existem ou não. O *Perl* criá-los-á consoante necessário, atribuindo aos elementos intermédios não utilizados `undef`.

Se se pretender acrescentar elementos no fim de uma linha, faz-se apenas

```
# adiciona colunas a uma linha existente
push @{ $AoA[0] }, "wilma", "betty";
```

Note que não se pode fazer

```
push $AoA[0], "wilma", "betty";    # ERRADO!
```

pois este código nem sequer compila. Porquê? Porque o argumento de `push()` tem de ser **mesmo** um array e não uma referência para um array.

## 9.3 Acesso e impressão

Imprimir apenas um dos elementos da estrutura é trivial:

```
print $AoA[0][0];
```

Se se pretender imprimir toda a estrutura, **não** se pode fazer

```
print @AoA;          # ERRADO!
```

pois apenas se obtém uma lista com as referências contidas no vector `@AoA`. Em vez disso, tem que se dereferir cada elemento sequencialmente para imprimir a estrutura completa:

```
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";      # os parêntesis rectos só embelezam a impressão
}
```

Se se pretender imprimir os índices, pode fazer-se:

```
for $i ( 0 .. $#AoA ) {
    print "\t elem $i is [ @{$AoA[$i]} ],\n";
}
```

ou mesmo (repare no ciclo interior em `$j`):

```
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. $#{$AoA[$i]} ) {
        print "\t elem $i $j is $AoA[$i][$j]\n";
    }
}
```

Isto está a complicar-se! É esta a razão porque, por vezes, é mais simples utilizar variáveis temporárias (uma referência `$aref`, neste caso).

```
for $i ( 0 .. $#AoA ) {
    $aref = $AoA[$i];
    for $j ( 0 .. $#{$aref} ) {
        print "\t elem $i $j is $AoA[$i][$j]\n";
    }
}
```

Hmm... ainda está feioso. E assim?

```
for $i ( 0 .. $#AoA ) {
    $aref = $AoA[$i];
    $n = @$aref - 1;
    for $j ( 0 .. $n ) {
        print "\t elem $i $j is $AoA[$i][$j]\n";
    }
}
```

## 9.4 Fatias

Se se pretende uma fatia (parte de uma linha) num array multidimensional tem de se recorrer a indexação mais ou menos complicada. Isto porque não há um equivalente, para fatias, à dereferência com apontadores que se aplica a elementos simples. (Lembre-se, claro, que pode sempre utilizar ciclos para obter as fatias.)

Eis como retirar uma fatia com um ciclo:

```
@fatia = ();
$x = 4;
for ($y = 7; $y < 13; $y++) {
    push @fatia, $AoA[$x][$y];
}
```

O ciclo podia ser substituído por um 'fatiamento' mais obscuro...

```
@fatia= @{ $AoA[4] } [ 7..12 ];
```

E se a fatia fôr bidimensional como, por exemplo quando `$x` varia de 4 a 8 e `$y` varia de 7 a 12? Eis uma forma simples...

```
@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
    }
}
```

Pode-se reduzir os ciclos com fatias...

```
for ($x = 4; $x <= 8; $x++) {
    push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];
}
```

Se trabalhasse com Transformadas Schwartzianas, possivelmente utilizava `map`...

```
@newAoA = map { [ @{ $AoA[$_] } [ 7..12 ] ] } 4 .. 8;
```

mas, se o seu chefe o acusasse de 'segurar' o emprego escrevendo código ilegível (só você o conseguia manter...), teria dificuldade em defender-se. Se fosse a si, escrevia uma função:

```
@newAoA = splice_2D( \@AoA, 4 => 8, 7 => 12 );
sub splice_2D {
    my $lrr = shift;                # lê ref para array com refs para array!
    my ($x_lo, $x_hi,
        $y_lo, $y_hi) = @_;         # lê os limites da indexação

    return map {                    # e volta a usar map 8-)
        [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
    } $x_lo .. $x_hi;
}
```

## 9.5 Arrays de hashes, hashes de hashes, etc...

Exemplifica-se brevemente alguns dos tópicos discutidos para os `@AoA` utilizando arrays de hashes, hashes de hashes, etc. A principal diferença, neste caso, vai ser na forma de aceder aos hashes: em vez de indexar com inteiros usa-se as funções `keys()` e `values()`.

Vamos definir um array de hashes (array que contém referências para hashes) explicitamente.

```
@AoH = (                            # cria array de referências para hashes
       { 1 => "fred", 2 => "barney" },
       { 1 => "george", 2 => "jane", 3 => "elroy" },
       { 1 => "homer", 2 => "marge", 3 => "bart" },
);
print $AoH[2]{'1'}, "\n";                 # imprime homer
```

Eis várias formas de imprimir os hashes apontados por cada referência no array:

```
for $aref ( @AoH ) {                # $aref refere sequencialmente cada hash
      @temp = %$aref;               # @temp é um vector temporário
      print "\t [ @temp ],\n";      # imprime todo o hash (em pares chave-valor)
      @temp = keys %$aref;
      print "\t [ @temp ],\n";      # imprime só as chaves
      @temp = values %$aref;
      print "\t [ @temp ],\n";      # imprime só os valores
}
```

Se se pretender explicitar os índices faz-se:

```
for $i ( 0 .. $#AoH ) {
    print "\t hash $i é [ @{[ %{$AoH[$i]} ]} ],\n";
}
```

Bem complicado né? `8-)`

Se se pretender imprimir elemento a elemento faz-se

```
for $i ( 0 .. $#AoH ) {
    for $j ( keys %{$AoH[$i]} ) {
        print "\t elem $i $j is $AoH[$i]{$j}\n";
    }
}
```

Esta foi mais legível.

Pode criar-se um `@AoH` implicitamente quando se faz, por exemplo, `$AoH[3]{'b'}=20;`

Para terminar mostra-se como é construído um `%HoH` (hash of hashes), quer na forma directa, quer utilizando uma referência, contendo os elementos (Simpsons) anteriormente já referidos.

Note que na construção da referência `$ref_to_HoH` os parêntesis exteriores são **chavetas** (devolvem uma referência para hash).

```
$ref_to_HoH = { a =>  { 1 => "fred", 2 => "barney" },
         b => { 1 => "george", 2 => "jane", 3 => "elroy" },
         c => { 1 => "homer", 2 => "marge", 3 => "bart" },
};

%HoH = { a =>  { 1 => "fred", 2 => "barney" },
         b => { 1 => "george", 2 => "jane", 3 => "elroy" },
         c => { 1 => "homer", 2 => "marge", 3 => "bart" },
};
print $HoH{b}{2}, "\n";               # imprime jane
print $ref_to_HoH->{c}{3}, "\n";      # imprime bart
```

Chega de estruturas de dados. Se quiser outras, vai ter de construí-las você mesmo!

# 10 Outras Funções em Perl

Apresenta-se uma lista parcial de funções do *Perl* ordenadas segundo um critério de aplicabilidade. Esta lista é um subconjunto da lista que consta da secção `perlfunc` do manual. Note que algumas funções se encontram repetidas. Note, também, que não se discute, nem utiliza neste documento, algumas das funções de uso mais raro ou mais específicas. Por exemplo, muitas delas estão intimamente ligadas ao núcleo do sistema operativo Unix/Linux, e esta introdução ao *Perl* não aborda esta matéria.

- **Funcões aplicáveis a $ESCALAREs ou strings**
  `chomp, chop, chr, crypt, hex, index, lc, lcfirst, length, oct, ord, pack, q/STRING/, qq/STRING/, reverse, rindex, sprintf, substr, tr///, uc, ucfirst, y///`
- **Expressões regulares e pesquisa de texto ('pattern matching')**
  `m//, pos, quotemeta, s///, split, study, tr//`
- **Funções numéricas**
  `abs, atan2, cos, exp, hex, int, log, oct, rand, sin, sqrt, srand`
- **Funções para @ARRAYs**
  `pop, push, shift, splice, unshift`
- **Funções para listas de dados**
  `grep, join, map, qw/STRING/, reverse, sort, unpack`
- **Funções para %HASHes**
  `delete, each, exists, keys, values`
- **Funções de entrada saída ('input/output')**
  `binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, rewinddir, seek, seekdir, select, syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write`
- **Funções para dados de dimensão fixa ou registos**
  `pack, read, syscall, sysread, syswrite, unpack, vec`
- **Funções para filehandles, ficheiros, ou directorias**
  `-X, chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, stat, symlink, umask, unlink, utime`
- **Termos relacionados com o controle da execução dos programas**
  `caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray`
- **Termos relacionados com o âmbito ('scope') das variáveis**
  `caller, import, local, my, package, use`
- **Funções várias**
  `defined, dump, eval, formline, local, my, reset, scalar, undef, wantarray`

- **Funções para processos e grupos de processos**
  `alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx/STRING/, setpgrp, setpriority, sleep, system, times, wait, waitpid`
- **Termos relacionados com módulos**
  `do, import, no, package, require, use`
- **Termos relacionados com classes e orientação para objectos**
  `bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use`
- **Funções relacionadas com tempo**
  `gmtime, localtime, time, times`

Da anterior lista, são aqui referidas mais algumas funções ainda não discutidas.

- **defined** `VARIABLE`
  Devolve verdadeiro se a variável `VARIABLE` existe e falso em caso contrário. Testa a existência de escalares, arrays e hashes.
- **exists** `EXPR`
  Dada uma expressão `EXPR` que especifica um elemento de um array ou um elemento de um hash, `exists` será verdadeira se o elemento especificado já foi inicializado, mesmo que o seu valor actual seja indefinido ('undefined'). O elemento não é 'auto-vivificado' se não existir.

  ```
  print "Existe\n"    if exists $hash{$key};
  print "Definido\n"   if defined $hash{$key};
  print "Verdade\n"      if $hash{$key};

  print "Existe\n"    if exists $array[$index];
  print "Definido\n"   if defined $array[$index];
  print "Verdade\n"      if $array[$index];
  ```

  Um elemento de um hash ou de um array é 'verdadeiro' só se estiver definido, e é definido só se existir, mas o inverso não é necessariamente verdade.

  Pode aplicar-se `exists` a subrotinas. Devolve 'verdadeiro' se a subrotina já foi declarada, mesmo que esteja indefinida. Mencionar o nome de uma subrotina no argumento de `exists` ou de `defined` não é o mesmo que declará-la.

  ```
  print "Existe\n"    if exists &subrotina;
  print "Definida\n"   if defined &subrotina;
  ```

  Finalmente, `EXPR` pode ser arbitrariamente complicada desde que a operação final seja a leitura de um array ou de um hash, ou o nome de uma subrotina.

  ```
  if (exists $ref->{A}->{B}->{$key})  { }
  if (exists $hash{A}{B}{$key})       { }
  if (exists $ref->{A}->{B}->[$ix])   { }
  if (exists $hash{A}{B}[$ix])        { }
  if (exists &{$ref->{A}{B}{$key}})   { }
  ```

Uma última nota: neste último exemplo, o elemento de nível mais profundo cuja existência se testa não é auto-vivificado, mas os elementos intermédios são-no. Assim, `$ref->{"A"}` e `$ref->{"A"}->{"B"}` irão ser criados devido ao teste de existência do elemento `$key`. Isto acontece sempre que o operador seta é utilizado.

- **binmode** `FILEHANDLE`
  Trata o ficheiro referido por `FILEHANDLE` como um ficheiro binário.

- **bless** `REF, [CLASSNAME]`
  'Abençoa' o item apontado pela referência `REF` tornando-o num objecto da classe `CLASSNAME` (ou da classe implícita na execução). A orientação por objectos em *Perl* é obscura `8-)`. Consulte os manuais relativos a OO em *Perl*.

- **getc** `FILEHANDLE`
  Devolve o próximo byte contíguo no ficheiro apontado por `FILEHANDLE`. No fim do ficheiro devolve a string nula (que dá falso em contexto lógico).

- **gmtime** `EXPR`
  Converte um tempo dado pela função `time` para uma data.

- **goto** `[LABEL|EXPR|&NAME]`

- **sleep** `EXPR`
  Esta função 'adormece' o script por `EXPR` segundos e devolve o número de segundos efectivamente passados, pois pode ser interrompida com um 'alarme'. Como a unidade de contagem de tempo é o segundo, não é suficientemente precisa para certas tarefas de temporização.

- **pack** `TEMPLATE, LISTA`
  Empacota a `LISTA` numa estrutura binária, segundo a sequência de caracteres definida em `TEMPLATE`, e devolve a string contendo a estrutura.

- **unpack** `TEMPLATE, EXPR`
  Faz a operação inversa de `pack`. Desempacota, para uma lista, a string especificada por `EXPR` na forma especificada por `TEMPLATE` e devolve-a.

# 11 Conclusão

Esta foi uma breve (?) introdução ao *Perl*. No futuro será expandida. Talvez seja convertida para `html`... "Who knows?" `8-)`.

Ficaram muitos assuntos por abordar. Não se esqueça que a actual edição (a terceira) do livro "Programming Perl", de Larry Wall et al., O'Reilly & Associates, tem cerca de 1100 páginas! Obviamente muitos aspectos do *Perl* tiveram que ser omitidos.

Finalmente, não se esqueça que a informação actualizada para a versão do *Perl* que utiliza está nas páginas de manual que normalmente são instaladas com o interpretador. Consulte-as!