Ficha 6 – Perl Avançado

Tópicos abordados:
- Funções
- Referências
- Recursividade
- Exercícios

Duração prevista: 1 aula

©2011: {rui.ferreira,patricio,leonel.santos,carlos.antunes,carlos.machado, nuno.gomes}@ ipleiria.pt

1 Funções em PERL

Como já foi visto nas fichas anteriores, o correto uso de funções permite uma melhor estruturação do código. De seguida explicitam-se boas práticas de uso das funções na linguagem PERL.

1.1. Definição de funções

Em PERL, uma função do utilizador é definida com recurso à seguinte sintaxe:

```
sub Funcao
{
  expressão_1;
  expressão_2;
(...)
  expressão_N;
}
```

Até agora não identificávamos nem a quantidade nem o tipo de parâmetros que uma função poderia receber. A partir de agora, sempre que a função receba um ou mais

parâmetros, vamos tornar a declaração **obrigatória**, pois tal permite que o interpretador PERL valide as chamadas às funções na parte que respeita aos parâmetros

De seguida vão ser apresentados alguns exemplos ilustrativos da declaração de funções.

Função com 1 parâmetro: escalar ou REFERÊNCIA (tratada no cap. 2)

Declaração:

```
sub Funcao($)
{
   # nota: ($) significa que a função vai receber
   # apenas 1 parâmetro do tipo escalar ou uma
   # referência para qualquer tipo de variável
}
```

Função com 4 parâmetros: 4 escalares ou 2 escalares e 2 REFERÊNCIAS, ou outra combinação com este tipo de variáveis

Declaração:

```
sub Funcao($$$$)
{
   # nota: significa que a função vai receber
   # parâmetros do tipo escalar ou referências para
   # qualquer tipo de variável por cada ($) existente
   # no protótipo da função
}
```

Função com 2 parâmetros: 1 vetor (REFERÊNCIA) e 1 escalar

Declaração:

```
sub Funcao($$)
{
   # nota: significa que a função vai receber
   # um parâmetro que é uma referência ($) para
   # um vetor e como segundo parâmetro um escalar ($)
}
```

Função com 3 parâmetros: 1 vetor (REFERÊNCIA), 1 escalar e 1 lista associativa (REFERÊNCIA)

Declaração:

```
sub Funcao($$$)
{
   # nota: significa que a função vai receber
   # 1 parâmetro que é uma referência ($) para
   # um vetor, como segundo parâmetro um escalar ($)
   # e como terceiro parâmetro uma referência ($)
   # para a lista associativa.
}
```

De seguida são apresentados mais alguns exemplos que mostram o modo de utilização das funções:

Nota: os exemplos recorrem ao conceito de referências que é explicado em detalhe na secção 2 desta ficha.

```
#!/usr/bin/perl
use strict;
use warnings; # sugere-se o uso de "use warnings" em vez do "-w"

sub funcao_1($$)
{
 my $ref_vetor = $_[0]; # obter referência do vetor
 $$ref_vetor[0] = 33; # modificar primeiro elemento do vetor

 $_[1] = 44;  # alterar valor do escalar (referência por omissão)

 my $scalar = $_[1];
 $scalar = 55;    # alteração não é visível na função principal
}

sub funcao_2($$$)
{
 my $ref_vetor = $_[0]; # obter referência do vetor
 $$ref_vetor[0] = 44;    # modificar primeiro elemento do vetor

 $_[1] = 66;  # alterar valor do escalar (referência por omissão)

 my $scalar = $_[1];
 $scalar = 77;    # alteração não é visível na função principal

 my $ref_hash = $_[2]; # obter referência da lista associativa
 $$ref_hash{"so"} = 99; # modificar a entrada com a chave 'so'
}


# (continua…)
```

```
# ++++++++++++  teste às funções  +++++++++++++++++
my $p1 = 1;
my @v = (1, 2, 3);
print("vetor inicial: @v\n");
print("escalar inicial: $p1\n");
funcao_1(\@v, $p1);
print("vetor final: @v\n");
print("escalar final: $p1\n");

print("\n");
@v = (1, 2, 3);
$p1 = 1;
my %ht = ('so' => 66);
print("vetor inicial: @v\n");
print("escalar inicial: $p1\n");
print("hash inicial: $ht{'so'}\n");
funcao_2(\@v, $p1, \%ht);
print("vetor final: @v\n");
print("escalar final: $p1\n");
print("hash final: $ht{'so'}\n");
```

1.2. Passagem de parâmetros

Lembrar que, no PERL, por omissão, a passagem de parâmetros é sempre feita por referência, ou seja, se alterarmos o valor apontados pelas referências dentro de uma função essas alterações tornam-se visíveis na função que fez a chamada. Deste modo, temos que ter alguns cuidados. Se não quisermos alterar os valores apontados pelas referências dentro de uma função apenas temos que fazer uma cópia dos parâmetros para uma variável local à função que estamos a utilizar.

No próximo exemplo existem duas funções que exemplificam este cenário:

- na primeira (**funcao_3($)**) utiliza-se diretamente o vetor (@_) para aceder e actualizar o primeiro elemento do vetor e, logo, a variável que foi passada para a função;
- na segunda (**funcao_4($)**) copia-se o valor do primeiro elemento do vetor para uma variável local e, como se pode verificar, ao actualizar essa variável local não se actualiza a variável da função principal que fez a chamada.

```
#!/usr/bin/perl
use strict;
use warnings; # sugere-se o uso de "use warnings" em vez do "-w"

# (continua…)
```

```
sub funcao_3($)
{
  $_[0] = 2;  # alterar valor do escalar (referência por omissão)
}

sub funcao_4($)
{
  my $i = $_[0]; #obter valor da variável passada por referência
  $i = 3;         # alteração não é visivel na função principal
}

# ++++++++++  teste às funções  +++++++++++++++
my $p1 = 1;
print("escalar inicial: $p1\n");    # $p1 = 1
funcao_3($p1);                    # há actualização da variável
print("escalar final: $p1\n");      # $p1 = 2

print("\n");
my $p2 = 1;
print("escalar inicial: $p2\n");    # $p2 = 1
funcao_4($p2);                 # não há actualização da variável
print("escalar final: $p2\n");      # $p2 = 1
```

2 Referências em PERL

Quando decidimos utilizar funções, para organizar melhor o código, deparamo-nos com um problema, isto claro, se precisarmos de passar mais de um tipo de parâmetros para a função. Assim, se apenas necessitarmos de passar apenas escalares ($) não vamos ter problemas, mas se incluirmos na chamada à função, por exemplo, escalares ($), vetores (@) e/ou listas associativas (%), aí podemos começar a ter alguns problemas. Para explicar melhor esta problemática vamos chamar funções com N parâmetros de vários tipos e ilustrar as situações problemáticas. Seguidamente mostrar-se-á uma forma, simples, de os resolver.

Função com 2 parâmetros escalares

nota: não existem problemas, pois os parâmetros são ambos escalares.

Chamada:

```
Funcao($p1, $p2);
```

Obter parâmetros na função:

```
my $fp1 = $_[0];     # 1° parâmetro
my $fp2 = $_[1];     # 2° parâmetro
```

Neste exemplo cada parâmetro vai para um índice no vetor (@_) ($fp1 a $_[0] e $fp2 a $_[1]), pelo que tudo funciona como o esperado.

Função com 2 parâmetros: primeiro um escalar e depois um vetor

nota: não existem problemas

Chamada:

```
Funcao($p1, @v1);
```

Obter parâmetros na função:

```
my $fp1 = $_[0];     # 1° parâmetro (escalar)
my $fv1 = $_[1];     # 2° parâmetro (início do vector)
```

Neste exemplo o escalar $p1 vai ter ao índice 0 do vetor (@_), e sempre a esse, enquanto o vetor @v1 vai ter como início o índice 1 do vetor (@_) e termina no índice N, sendo N o tamanho do vetor @fv1.

A chamada à função com esta ordem de parâmetros funciona como o esperado.

Função com 3 parâmetros: um vetor, um escalar e uma lista associativa

nota: existem problemas, pois não é possível saber onde termina o primeiro vetor e
começa o escalar e/ou o próxima lista associativa.

Chamada:

```
Funcao(@v1, $p1, %ht1);
```

Obter parâmetros na função:

```
my @fv1 = $_[0];     # 1º parâmetro (início do vector)
my $fp1 = $_[???];   # ??? qual é o índice ???
my %fht2 = $_[???]; # ??? qual é o índice ???
```

Neste exemplo o vetor @fv1 vai ter como início o índice 0 do vetor (@_). Mas depois, dentro da função, visto que o vetor (@_) conter todos os parâmetros numa única estrutura, não é possível saber onde estão localizados os próximos parâmetros. E no caso da lista associativa, como será que se poderia obter os valores associados a essa lista?

Para resolver os problemas que podem surgir das inúmeras combinações que se podem fazer com base no exemplo apresentado anteriormente, aconselha-se a **passagem de parâmetros por referência**.

2.1 Passagem de parâmetros por referência

Poderá criar referências para uma variável, função ou valor usando o carater barra invertida ('\'). Esta barra funciona de modo similar ao operador & (endereço de) da linguagem C. Aqui estão alguns exemplos:

```
$ref_scalar = \$foo;        # \$ representa referência
$ref_array  = \@ARGV;       # \@ representa referência
$ref_hash   = \%ENV;        # \% representa referência
```

Importa referir que as referências são *tipadas*, isto é, estão associadas a um determinado tipo de estrutura:SCALAR para escalares, ARRAY para vetores e HASH para listas associativas. O tipo de uma referência é indicado pela função *print* quando se usa uma referência. Assim, A listagem seguinte ilustra o conceito (no final da listagem está a saída que a execução do script produz).

```
#!/usr/bin/perl
use strict;
use warnings;
# (continua…)
```

```
my $Examples = "Exemplo";
my $ref_scalar = \$Examples; # ref. para "scalar"
my $ref_array  = \@ARGV;      # ref. para vetor @ARGV
my $ref_hash   = \%ENV;      # ref. para %ENV
print("\$ref_scalar = '$ref_scalar'\n");
print("\$ref_array = '$ref_array'\n");
print("\$ref_hash = '$ref_hash'\n");

#+++++++++++++ Saída da script ++++++++++++++
$ref_scalar = 'SCALAR(0x104a16c8)'
$ref_array = 'ARRAY(0x104a1020)'
$ref_hash = 'HASH(0x104a1098)'
```

O seguinte quadro demonstra o acesso direto aos valores dos vários tipos de variáveis passadas por referência:

```
# aceder ao valor de escalares
 print("$$ref_scalar \n");

# aceder ao valores de vetores
 print("$#$ref_vetor \n");         # último índice do vetor
 print("@$ref_vetor \n");          # vetor
 print("$$ref_vetor[0] \n");       # índice 0 do vetor

# aceder aos valores de listas associativas
 print("$$ref_hash{'so'} \n");     # valor da chave 'so'

 my @chaves = keys(%$ref_hash);    # obter chaves da lista
 print("@chaves \n");

 my @valores = values(%$ref_hash); # obter valores da lista
 print("@valores \n");
```

2.2 Uso de referências em escalares

```
#!/usr/bin/perl
use strict;
use warnings; # sugere-se o uso de "use warnings" em vez do "-w"

# Função que recebe a referência para um valor escalar,
# faz o incremento de 1 unidade actualiza esse valor
# nota: utilização direta da referência $_[0]
sub myIncrementa1($)
{
  ${$_[0]}++;  # referência com conversão explicita {}
}


# (continua…)
```

```
# Função que recebe a referência para um valor escalar,
# faz o incremento de 2 unidades actualiza esse valor
# nota: utilização de uma referência auxiliar
sub myIncrementa2($)
{
  my $ref_scalar = $_[0];
  $$ref_scalar += 2;
}

# ++++++++++  teste às funções  +++++++++++++++
my $digito = 0;

print("Antes da função: $digito \n");

myIncrementa1(\$digito);     # \$ representa referência
print("Depois da função myIncrementa1: $digito \n");

myIncrementa2(\$digito);     # \$ representa referência
print("Depois da função myIncrementa2: $digito \n");
```

2.3 Uso de referências em vetores

```
#!/usr/bin/perl
use strict;
use warnings; # sugere-se o uso de "use warnings" em vez do "-w"

# Função que recebe uma referência para um vetor e
# o número de elementos a "modificar" (coloca a
# primeira letra a maiúscula)
# nota: utilização de referências auxiliares
sub myUcFirst($$)
{
  my $ref_array = $_[0];  # referência para o vetor
  my $num_elems = $_[1];  # obter número de elementos
  my $i;

  # verificar se o número de elementos a modificar
  # é superior ao número de elementos do vetor
  if ($num_elems > @$ref_array)
  {
     $num_elems = @$ref_array;
  }

  # modificar os $num_elem primeiros elementos do vetor
  # exemplo com a utilização de ($$) e (->),
  # que são formas equivalentes
  for ($i = 0; $i < $num_elems; $i++)
  {
     $$ref_array[$i] = ucfirst($$ref_array[$i]);
  }
}

# (continua…)
```

```
# ++++++++++++  teste à função  +++++++++++++++++
my $quant = 3;
my @numeros = ("um","dois","tres","quatro","cinco");

print("Initial Array: @numeros \n");

myUcFirst(\@numeros, $quant);# \@ representa referência

print("Final Array: @numeros \n");
```

2.4 Uso de referências em listas associativas (*hashes*)

```
#!/usr/bin/perl
use strict;
use warnings; # sugere-se o uso de "use warnings" em vez do "-w"

# Nota: atenção às diversas formas como se
# podem utilizar as referências passadas às funções

# Função que recebe uma lista associativa (por
# referência) e dois escalares (para adicionar à lista
# associativa)

sub AddToHash($$$)
{
  my $ref_hash = $_[0]; # referência da lista associativa
  my $key = $_[1];      # obter chave (p/valor)
  my $value = $_[2];    # obter valor (p/valor)

  $$ref_hash{$key} = $value;  # adicionar à lista
}

# Função que mostra uma lista associativa (referência)
sub ShowHash($)
{
  my $ref_hash = $_[0];   #referência da lista associativa
  my $codigo;

  foreach $codigo (keys(%$ref_hash))
  {
     print("$codigo - $$ref_hash{$codigo} \n");
  }
}

# ++++++++++++  teste às funções  +++++++++++++++++
my %codPostal = ('2400', 'Leiria', '3000', 'Coimbra');
my $cod = '2000';
my $area = 'Porto';
# mostrar a lista inicial
print("Lista inicial:\n");
ShowHash(\%codPostal);

# (continua…)
```

```
# adicionar novo par à lista
# 1º parâmetro passado por referência
# 2º e 3º parâmetros passados por valor
AddToHash(\%codPostal, $cod, $area);
print("\n");

# mostrar lista final, depois de ser alterada
print("Lista final:\n");
ShowHash(\%codPostal);
```

Exercício 1

Escreva o programa "decompoe_hash.pl", que crie uma lista associativa com base nos meses e respetivos número de dias. O programa deverá assentar em três funções, a seguir indicadas:

1) Função para mostrar, de forma personalizada, a lista associativa.
2) Função que receba a lista associativa e dois vetores e que devolva um dos vetores com as chaves e o outro com os valores da lista associativa;
3) Função para mostrar, de forma personalizada, o conteúdo dos vetores;

Utilize as funções para mostrar a lista e os vetores que são utilizados no programa.

2.5 Referências anónimas

O PERL suporta o conceito de referência anónima. Uma referência anónima permite a criação de dados anónimos, no sentido que os dados não ficam associados a nenhuma variável, mas sim a uma referência. O uso de referências anónimas permite, por exemplo, que sejam guardados listas associativas (via referência anónima) em cada elemento de um vetor, ou um vetor como valor dos elementos de uma lista associativa. As referências anónimas são criadas de formas diferentes consoante se trate de referências anónimas para vetores ou para listas associativas.

2.5.1 Referências anónimas para vetores

Um vetor é anonimizado através do uso de parêntesis retos. Atenda-se à listagem seguinte na qual uma lista associativa (**%Linhas_H**) guarda as características técnicas de várias máquinas. A lista associativa é indexada pelo nome da máquina (e.g, "Maquina 1"), sendo que as caraterísticas técnicas são guardadas num vetor de strings, vetor esse que é registado na lista associativa via referência anónima.

```
#!/usr/bin/perl
use strict;
use warnings;
my %Linhas_H;

# Definição de máquinas atráves de vetores anónimos
$Linhas_H{"Maquina 1"} = ["Core 2 Duo", "2.0 GHz", "2
 GB RAM","250 GB Disco"];

$Linhas_H{"Maquina 2"} = ["intel i7", "3.0 GHz", "8 GB
 RAM","1 TB Disco"];

# Acrescenta "GPU" à definição de cada
# máquina, mostrando o vetor anónimo associado
# a cada chave da lista associativa
foreach my $MachName_S ( keys(%Linhas_H) )
{
  my $Ref_Computer = $Linhas_H{$MachName_S};

  # A string "GPU" é acrescentada ao vetor
  @$Ref_Computer =  (@$Ref_Computer, "GPU");

  print("'\$Ref_Computer = $Ref_Computer\n");
  print("'$MachName_S' = '@$Ref_Computer'\n");
}
```

2.5.2 Referências anónimas para listas associativas

A obtenção de uma referência anónima para uma lista associativa segue um processo análogo ao da criação de referência anónima para vetor, com exceção do uso de chavetas. Atenda-se à listagem seguinte que apresenta duas formas distintas de criar referências anónimas para listas associativas.

```
#!/usr/bin/perl
use strict;
use warnings;

# the vector will hold one anonymous reference
# to hash per cell
my @Estudantes_L = ();

# [0] holds a reference to an anonymous hash
$Estudantes_L[0] =
{
    'nome'=>'Super Mario',
    'anoNascimento'=> '1999',
    'curso'=>'EI'
};


# (continua…)
```

```
# Another anonymous references to hash
$Estudantes_L[1] =
{
    'nome'=>'Super Futre',
    'anoNascimento'=> '1966',
    'curso'=>'Futebol'
};

# Another one, this time with a different
# definition format for the hash
$Estudantes_L[2] =
{
    ('nome','Rafael Nadal',
         'anoNascimento','1986','curso','tenis')
};


my $i = 0;
foreach my $ref_Estudante (@Estudantes_L)
{
    $i++;
    print("[$i]\n");
    foreach my $Key_S(sort(keys(%$ref_Estudante)))
    {
         print("$Key_S='$$ref_Estudante{$Key_S}'\n");
    }
}
```

3 Recursividade

Em muitas linguagens de programação, os programas são estruturados com funções que se chamam de forma hierárquica (uma função chama a outra e assim por diante). Para algumas situações é útil que uma função volte a chamar-se a ela própria (no contexto de uma nova instância) antes de retornar um valor. As funções que chamam novas instâncias delas próprias designam-se por funções recursivas.

Uma função recursiva é, pois, uma função que chama uma nova instância de si própria antes de terminar a instância corrente. O processo de chamar novas instâncias da mesma função **repete-se até que se alcance a chamada condição de paragem** da recursividade.

3.1 Fatorial como exemplo de recursividade

Um exemplo clássico de recursividade é o cálculo do fatorial de um número. Como sabe, o fatorial de um número inteiro n, designa-se pela simbologia **n!** e é calculado como o produtório dos números inteiros de 1 até n. Deste modo, o fatorial de n (n!) é igual a:

n! = 1 * 2 * ... * (n-1) * n (equação 1)

Por exemplo, o fatorial de 5 corresponde a 1*2*3*4*5 = 120.

Facilmente se depreende que o fatorial de n pode ser calculado a partir do fatorial de n-1, ou seja:

n! = (n-1)! * n (equação 2)

Exemplificando, tem-se que:

5! = 4! * 5 e que 4! = 3! * 4 e assim sucessivamente.

Isto significa que 5! pode ser calculado com base em 4! e que por sua vez, 4! pode ser calculado com base em 3! e assim sucessivamente. Tem-se pois um cálculo recursivo.

3.2 Cálculo do fatorial em PERL

O cálculo do fatorial de n pode ser realizado por duas vias: forma iterativa e forma recursiva. A primeira recorre a um ciclo com n iterações, ao passo que a forma recursiva assenta, como o nome sugere, na recursividade.

```
sub fatorial_iterativo($)
{
  my $n = $_[0];
  my $fatorial = 1;
  my $i;
  foreach $i (1..$n)
  {
    $fatorial *= $i;
  }
  return $fatorial;
}
```

```
sub fatorial_recursivo($)
{
  my $n = $_[0]; # obter parâmetro de entrada
  if($n <= 1)    #condição de paragem
  {
     return 1;
  }
  else
  {
     # chamada recursiva
     return  $n * fatorial_recursivo($n-1);
  }
}
```

A figura que se segue exemplifica o cálculo via função recursiva de **5! = 5*4*3*2*1**:

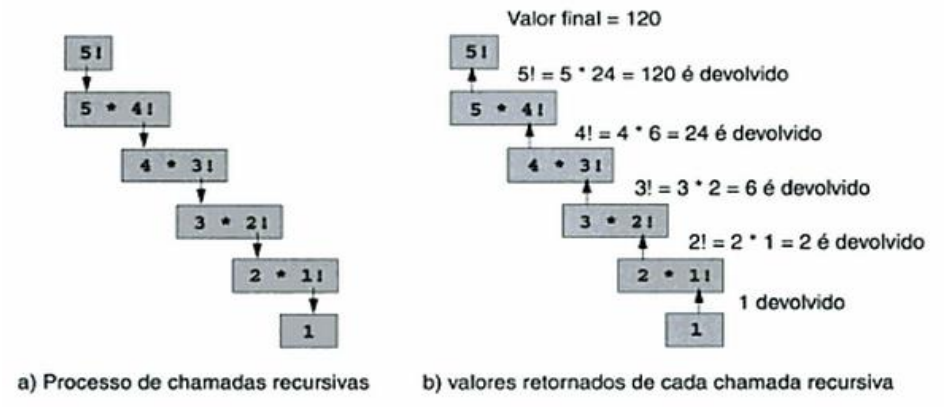

Figura 1

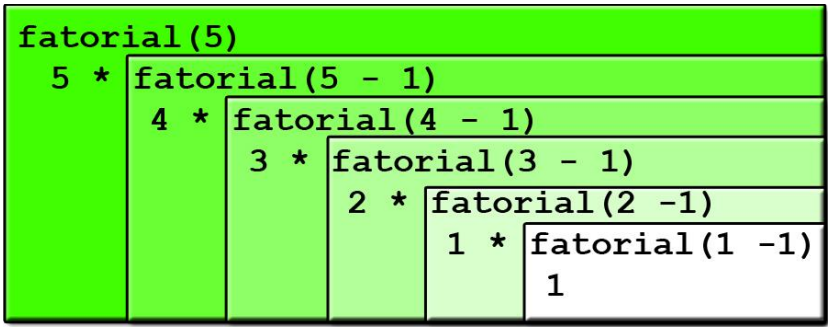

Figura 2 - (fonte: http://olamundo0.wordpress.com/2010/04/20/recursividade/)

Exercício 2

Construa um *script* em PERL, "fatorial.pl", que deverá receber por argumentos o limite superior e inferior de valores, e apresentar os factoriais de todos os valores inteiros compreendidos no intervalo. Deve, também, criar uma função para verificar se os argumentos são numéricos (inteiros positivos) e validar se o primeiro argumento é menor ou igual ao segundo.

Exemplo:

```
fatorial.pl  2 4
```

Saída:

```
2! = 2*1 = 2;
3! = 3*2*1 = 6;
4! = 4*3*2*1 = 24;
```

Exercício 3

Elabore a script "pesquisaMultimedia2.pl", cujo propósito é o de criar uma lista ordenada, com o nome dos ficheiros multimédia (.avi, .wmv, .mp3) existente num diretório e respetivos subdiretórios do diretório actual.

Nota: Faça uso da metodologia *readdir* e da metodologia *glob*.

Exercício 4

Efectue uma pesquisa na Web para que possa encontrar informação (que não seja código) para implementar uma solução (recursiva) para os "Números de Fibonacci".

4 Exercícios

1) Elabore o script refARGV.pl que deve implementar a função ARGVtoStr($) que recebendo uma referência para o vetor de parâmetros @ARGV, deve devolver uma string que contenha todos os argumentos existentes no vetor @ARGV. Cada parâmetro deve ser separada por um "\n".

2) Elabore o script refENV.pl que deve implementar a função ENVtoStr($) que recebendo uma referência para a lista associativa %ENV, deve devolver uma string que contenha todas as variáveis do ambiente e respetivos valores existentes na lista associativa %ENV. A string devolver deve conter cada variável do ambiente (e respetivo valor) em linhas separadas.

3) Recorrendo à linguagem PERL, elabore o script **linhasDasPalavras.pl** cujo propósito é o de identificar as palavras existentes num ficheiro de texto, registando para cada uma das palavras a(s) linha(s) onde ocorre. O script deve recorrer a uma lista associativa (*hash*), usando palavra como chave e como valor, uma referência para vetor que regista o número de cada linha onde a palavra existe. No final, o script deve mostrar na saída padrão uma listagem em que cada linha se inicia por uma palavra seguida pelo números das linhas em que ocorre. O nome do ficheiro deve ser passado como único argumento da linha de comando. O script deve assentar em duas funções: FileToListagem que processa o ficheiro, devolvendo a lista associativa acima descrita e ListagemToStdout que deve mostrar a listagem (representada pela *hash*) na saída padrão. As funções bem como os respetivos parâmetros devem seguir as recomendações definidas nesta ficha prática, sendo que estruturas como listas associativas devem ser passadas por referência.

   NOTA: faça uso de referências anónimas.

4)

O máximo divisor comum de dois números é, como o nome sugere, o maior número inteiro que divide ambos os números sem deixar resto. Elabore o script mdc.pl, que assente em recursividade, determina o máximo divisor comum de dois números que são pedidos ao utilizador via entrada padrão.

http://www.ualberta.ca/~hquamen/303/recursion.html

5) O cálculo da potência de um número envolve a multiplicação repetida desse número (a base) um determinado número de vezes (o expoente). Elabore o script potencia.pl, que implementa uma versão recursiva do cálculo da potência, em que o valor da base e do expoente são pedidos ao utilizador via entrada padrão.

6) Elabore o script soma_r.pl que deve implementar uma versão recursiva da soma de uma série de números guardados num vetor. Os números são pedidos ao utilizador via entrada padrão, até este indicar o valor zero.

   NOTA: Faça uso de referências.

7) Elabore o script tree.pl cuja funcionalidade é apresentar a estrutura hierárquica das sub-diretorias de uma diretoria cujo caminho é passado como parâmetro da linha de comandos. Por exemplo, se o script for executado da seguinte forma:

```
$ ./tree.pl /usr/share/
```

O script deve produzir o seguinte resultado:

```
/usr/share/
    |-foo2lava
    |---icm
    |-nautilus
    |---icons
    |-----hicolor
    |-------48x48
    |---------emblems
    |-------16x16
    |---------emblems
    |-------24x24
    |---------emblems
    |---patterns
    |---ui
    |-gst-python
    |---0.10
    |-----defs
    |-xsessions
      ...
```

8) Elabore o script permutacoes.pl que deve apresentar no ecrã as permutações de uma *string*, cujo valor é pedido ao utilizador via entrada padrão. Por exemplo, se o utilizador indicar o seguinte texto:

```
Indique uma palavra: ABC
```

O script deve produzir o seguinte resultado:

```
ABC
BAC
BCA
ACB
CAB
CBA
```

9) Como podem ser implementadas estruturas de árvores binárias em PERL?

NOTA: pode consultar o wikipedia a respeito de árvores binárias (http://en.wikipedia.org/wiki/Binary_tree).

5 Bibliografia

"Modern PERL", Chromatic, **ISBN-10:** 0-9779201-5-1. Disponível em http://www.onyxneon.com/books/modern_perl/index.html.

"Beginning PERL – chapter 7: References", Simon Cozens, 2000. Disponível em http://learn.perl.org/books/beginning-perl/.