Resolução de alguns exercícios da ficha 3 e exemplos adicionais

1. Utilizando programação em Bash, construa uma *script* "compara" que recebe dois inteiros e mostra qual a relação existente (igual, maior, menor) entre eles. O resultado do comando deverá ser compatível com o seguinte exemplo:

```
$ compara 2 3
Eu sou o comando compara e 2 e menor que 3

$ compara 5 3
Eu sou o comando compara e 5 e maior que 3

$ compara 7 7
Eu sou o comando compara e 7 e igual a 7
```

Uma resolução possível

```
#!/bin/bash

if [ $# -ne 2 ]        # ou então   if test $# -ne 2
then
   echo erro nos parametros
   echo forma de usar
   echo $0 numero1 numero2
else
   if [ $1 -lt $2 ]   # ou então   if test $1 –lt $2
   then
      echo "eu sou o comando $0 e $1 e menor que $2"
   elif [ $1 -gt $2 ]
   then
      echo "eu sou o comando $0 e $1 e maior que $2"
   else
      echo "eu sou o comando $0 e $1 e igual a $2"
   fi
fi
```

A primeira linha é, naturalmente, a indicação de *script* (#!) seguida da indicação de qual o interpretador de comandos a usar para interpretar a script (/bin/bash). Todas as *scripts* neste documento terão esta linha. A *script* começa por usar uma estrutura *if* para verificar se o número de argumentos fornecidos na linha de comandos (indicado pela variável $#) é 2. Se não for (o operador -ne significa *not equal*), é indicada uma mensagem de erro e a forma de usar a script.

Caso o número de argumentos seja 2, através de uma nova estrutura *if* a script averigua se o primeiro argumento ($1) é menor (operador –lt) que o segundo ($2). Se for é apresentada a mensagem apropriada através do comando echo. Caso contrário, através do uso de *elif*, é sucessivamente testada a hipótese de o primeiro argumento ser maior que o segundo, e finalmente a hipótese de serem iguais.

2. Construa a script “tipo” que identifica o tipo do ficheiro dado como argumento (dois casos: “directoria” ou “ficheiro”), e no caso de ser “ficheiro”, se tem a propriedade de “executável”. Deve-se prever o caso em que o ficheiro não existe. O resultado do comando deverá ser consistente com o exemplo seguinte (assumindo que “aula4” existe e é uma directoria, aula4.txt é um ficheiro regular não executável, ambos na directoria onde a script é executada):

```
$ tipo aula4
aula4 ---> directoria executavel

$ tipo aula4/aula4.txt
aula4/aula4.txt ---> ficheiro

$ tipo /bin/ps
/bin/ps ---> ficheiro executavel

$ tipo /bin/ttt
[Erro] /bin/ttt nao existe!
```

Uma resolução possível

```
#!/bin/bash

if [ $# -ne 1 ]              # ou então if test $# -ne 1
then
   echo erro nos parametros
   echo $0 ficheiro
else
   if test -e "$1"           #  ou então  if [ -e "$1" ]
   then
      if test -f "$1"        #  ou então  if [ -f "$1" ]
      then
         echo "$1 e um ficheiro regular"
         if test –x "$1"
         then
            echo "$1 e executavel"
         fi
      elif test -d "$1"      #  ou então  if [ -d "$1" ]
      then
         echo "$1 e uma directoria"
      else
         echo "$1 e um tipo de ficheiro não suportado"
      fi
   else
      echo "$1 nao existe"
   fi
fi
```

Esta *script* começa por testar se foi fornecido um argumento através de uma estrutura *if*, usando a variável $# que indica o número de argumentos fornecidos. Recordar que *[etc]* é equivalente a *test etc*.

De seguida, a script utiliza várias estruturas *if* encadeadas uma nas outras para testar sucessivamente a existência do ficheiros, e existindo o ficheiro, verificar se se trata de um ficheiro regular, uma directoria, ou outro tipo (não suportado). No caso de ser um ficheiro regular, é averiguada adicionalmente a hipótese de ser executável. Todos os testes ao ficheiro são efectuados através do comando *test* (o qual

poderia também ser escrito usando [ ]). O comando *test*, tal como indicado nos resumos fornecidos e documentação disponibilizada pelo comando *man*, permite testar directamente várias propriedades acerca de um ficheiro: se existe (-e), se é um ficheiro regular (-f), se é uma directoria (-d) se é executável (-x). Existem outros testes possíveis, descritos na documentação fornecida e nas páginas de manual (comando *man*).

Neste exemplo deve ser dada particular atenção à sintaxe das estruturas *if*, em particular quando encadeadas umas nas outras. Reparar como e quando o *fi* é usado para terminar um *if*.

3. Utilizando programação em Bash, construa uma *script* "pot2" que apresenta as potências de 2 de 0 até 10. O resultado do comando deverá ser consistente com o exemplo de utilização dado abaixo:

```
$ pot2
2^0 = 1
2^1 = 2
2^2 = 4
...
2^10 = 1024
```

Uma resolução possível

```
#!/bin/bash

exp=0
pot=1
while test $exp -le 10       # ou então while [ $exp –le 10 ]
do
   echo "2^$exp = $pot"
   exp=$((exp+1))
   pot=$((pot*2))
done
```

Esta solução utiliza um ciclo para percorrer as potências de valores 2^n, com *n* a variar entre 0 e 10. Para este efeito, a script começa com o valor de 2^0, multiplicando esse valor sucessivamente por 2, obtendo sucessivamente 2^1, 2^2, 2^3, etc. até 2^{10}. O valor de 2^n (com *n* entre 0 e 10) é mantido na variável *pot*. O expoente (o valor que vai de 0 a 10) é mantido na variável *exp*. Esta última variável é usada para controlar o ciclo *while*.

Nesta solução é de reparar nos seguintes aspectos

- A forma como se escrevem expressões aritméticas e a obtemnlção do valor resultante dessas expressões. Para tal usa-se a sintaxe $((…))
- A forma como se actualizam as variáveis *pot* e *exp*. Em ambos os casos faz-se a variável tomar o resultado de uma expressão aritmética usando a sintaxe mencionada no ponto anterior.
- O ciclo *while* também poderia ser expresso como um ciclo *for*:
    - Usando a forma *for ((…))* : ficava → for ((exp=0; exp <= 10; exp++))
    - Usando a forma *for variável in lista*
        - Indicando os valores explicitamente: ficava → *for exp in 0 1 2 3 4 5 6 7 8 9 10*
        - Usando o comando seq para gerar a sequência: ficava → *for exp in $(seq 0 10)*

  Usando um ciclo *for*, a actualização exp=$((exp+1)) deverá ser removida.

Uma outra resolução possível

```
#!/bin/bash
in/bash

for ((i=1; i<=10; i++))          # ou então:  for i in 1 2 3 4 5 6 7 8 9 10
do
  echo "2^$i = $(( 2 ** i))"
done
```

Nesta resolução usa-se um ciclo *for* e apresenta-se o resultado recorrendo ao operador ** (exponenciação), não sendo necessário estar a guardar uma variável *pot* com os valores sucessivos.

4. Crie um script em Bash que permita calcular o factorial de um número. Para esse efeito, deverá receber da linha de comandos o número cujo factorial se pretende calcular e apresentar no monitor o valor do factorial, tal como se mostra no exemplo abaixo.

```
$ ./factorial 8
-------------------------
8! = 40320
-------------------------

$ ./factorial -2
-------------------------
Argumento invalido!!!
-------------------------
```

Resolução

```
#!/bin/bash

if [ $# -ne 1 ]
then
   echo "erro nos parametros"
else
   fact=1;
   for ((i=2; i<=$1; i++))
   do
      fact=$(( $fact * $i ))
   done
   echo "$1! = $fact"
fi
```

Esta resolução não apresenta nada de novo em relação aos exercícios anteriores. Usa-se um ciclo *for* para fazer percorrer uma variável de 2 até ao número *n* fornecido como argumento. Em cada iteração multiplica-se uma variável que armazena o factorial (*fact*) pelos vários valores de 2 até *n*. Após a última iteração, o valor existente na variável *fact* tem o valor do factorial. Apesar de não ser estritamente necessário o uso de $ nas variáveis dentro da estrutura sintáctica $(( )) , optou-se por manter o seu uso (variáveis *fact* e *i*) para ser mais visível o facto de se tratarem de variáveis.

5. Construa a *script* utilizadores, que permita visualizar todos os utilizadores do sistema organizados pelo grupo primário a que pertencem. O resultado do comando deverá seguir o formato do exemplo abaixo: os grupos são listados por ordem alfabética, em letras maiúsculas, e, para cada grupo apresentado, é indicada a lista de utilizadores que têm esse grupo como grupo primário.

```
$ utilizadores
SOSD
joao
pedro

TI
antonio
maria
jose
```

Uma resolução possível

```
#!/bin/bash

cat /etc/group |
while read linha
do
   grupo=$(echo "$linha" | cut -d":" -f1 | tr "a-z" "A-Z")
   gid=$(echo "$linha" | cut -d":" -f3)
   echo "$grupo ($gid)"

   cat /etc/passwd |
   while read lll
   do
      login=$(echo "$lll" | cut -d":" -f1)    # é “LLL” e não “cento-e-onze”
      gnum=$(echo "$lll" | cut -d":" -f4)
      if test $gnum -eq $gid
      then
         echo "$login"
      fi
   done

   echo ""
done
```

O facto de se pretender controlar a apresentação pelo nome do grupo, sendo os utilizadores apresentados de forma subordinada a cada grupo vai obrigar à seguinte estratégia: percorre-se o ficheiro dos grupos (/etc/group) linha a linha (ou seja, grupo a grupo). Para cada grupo nesse ficheiro, vai-se percorrer o ficheiro dos utilizadores (/etc/passwd) e indicar todos os utilizadores que tem como grupo primário o grupo de /etc/group que se está actualmente a percorrer. Há ainda que ter em atenção ao facto de que no ficheiro dos utilizadores (/etc/passwd), a identificação do grupo primário é dada de forma numérica, sendo necessário obter esse valor no ficheiro dos grupos.

A estratégia algorítmica pode ser descrita em pseudo-código da seguinte forma:

Para cada linha de /etc/group (ou seja, para cada grupo no sistema) faz
Obtém o nome do grupo
Obtém o número do grupo
Para cada linha de /etc/passwd (ou seja, para cada utilizador) faz
Obtém o username
Obtém o número do grupo primário
Se o grupo primário do utilizador for igual ao do grupo de /etc/passwd então faz
Imprime o username do utilizador

Nota: esta estratégia obriga a percorrer o ficheiro dos utilizadores (/etc/passwd) na totalidade para cada grupo, tornando-o algo lento. Existem alternativas mais rápidas.

Nesta solução é de reparar nos seguintes aspectos:

- Percorrer um ficheiro linha a linha é feito usando um read como condição de um while.
  - cat /etc/group |
  - while read linha

  O comando *read* retorna o valor *true* enquanto não chegar ao fim do texto de onde está a ser obtida a informação lida, fazendo o ciclo executar mais uma iteração até se esgotar o texto a ser lido. A fonte de dados para o *read* é o output do comando *cat* que precede o *while*. De facto, o uso do redireccionamento com *pipe* no comando *cat* envia o output do *cat* para o comando seguinte, que neste caso é o *read*, isto apesar de haver um *while* pelo meio (recordar que o *while* não é um comando mas sim uma funcionalidade interna à bash, tornando-o "transparente" na questão do redireccionamento entre o *cat* e o *read*.
- A mesma estratégia é usada tanto para percorrer o ficheiro /etc/group como para o ficheiro /etc/passwd
- O nome e o número do grupo são obtidos e armazenados em variáveis com recurso à sintaxe $(( )) . Esta sintaxe permite executa um ou mais comandos, sendo o output apanhado e, neste caso, armazenado nas variávels grupo e gid:
  - grupo=$(echo "$linha" | cut -d":" -f1 | tr "a-z" "A-Z") → Obtém o nome do grupo, tendo já as letras minúsculas sido substituídas por maiúsculas (comando tr)
  - gid=$(echo "$linha" | cut -d":" -f3) → Obtém o IF (número) do grupo
- A obtenção do username e do número do grupo primário no ficheiro de utilizadores segue uma estratégia semelhante.
- Também se poderia ter usado a sintaxe ` ` em vez de $(( ))

Exemplos adicionais

A. A script seguinte permite copiar para a directoria backup todos os ficheiros que tenham mais do que 100 caracteres.

```
#!/bin/bash

mkdir backup

for fich in *
do
   if test -f $fich
   then
      tam=$(cat $fich | wc -c)
      if test $tam -ge 100
      then
         cp $fich backup
         echo "$fich ($tam)"
      fi
   fi
done
```

Acerca desta *script* há apenas a salientar os seguintes aspectos:

- Uso da sintaxe *for variável in lista*. Neste caso, a lista é simplesmente o asterisco, o que automaticamente "expande" para a lista dos nomes de todos os ficheiros na directoria actual. Desta forma consegue-se facilmente ter um ciclo que percorre cada um dos ficheiros existentes.
- A obtenção do tamanho do ficheiro é feita com recurso à seguinte ideia: imprime-se o conteúdo do ficheiro, redireccionando este para o comando wc, ao qual se diz para contar o número de caracteres recebidos. O efeito final é a obtenção do número de caracteres existentes no ficheiro. Através da sintaxe $(( )) consegue-se armazenar o output (o número de caracteres) numa variável (também se poderia usar a sintaxe ` `).

B. A *script* seguinte consegue adivinhar o número em que o utilizador pensou, desde que esteja entre 1 e 100, e que o utilizador vá respondendo honestamente se o número indicado pelo computador é maior, menor, ou igual àquele que foi pensado. Esta *script* não é um exemplo típico de *scripts*: normalmente as *scripts* cumprem tarefas de administração (tal como o exemplo anterior). No entanto, trata-se de mais um exemplo daquilo que é possível fazer com a sintaxe das *scripts*, e de mais um auxiliar de estudo, pelo que se inclui aqui.

```
#!/bin/bash

echo               # isto e um comentario
echo pensa num numero entre 1 e 100. vou adivinha-lo
echo
min=1; max=100
conta=0
resposta=nada;
while [ "$resposta" != "sim" ]   # aspas recomendadas. espaco em [] necessarios
do
   num=$(( ($min+$max)/2 ))
```

```
   conta=$(( $conta + 1 ))
   echo e $num \?
   read resposta
   if [ "$resposta" == "sim" ]; then        # reparar no ;
      echo acertei em $conta tentativas
   elif [ "$resposta" == "menor" ]; then    # novamente o ;
      echo ok, então vou dizer um numero menos que $num
      max=$num
   elif [ "$resposta" == "maior" ]; then
      echo ok, então vou dizer um numero maior que $num
      min=$num
   else
      echo não percebi o que escreveste
   fi
done
```

Não existe nada de novo nesta *script* em relação às anteriores, excepto o tema em si. Acerca desta *script* há apenas a salientar os seguintes aspectos:

- A forma como se pode escrever uma estrutura if (com ou *elif* e *else*). Notar a forma como cada fi fecha um *if* (os *else* não são fechados poro si só). Notar também que o *then* é obrigatório sempre que há uma condição, ou seja, nos *if* e nos *elif*.
- O uso de “;” para separar comandos separados que por acaso estão na mesma linha. O *then* na mesma linha que o *if* é um caso (mas há mais no exemplo) em que o “;” se torna necessário.
- O uso de expressões aritméticas e o uso da sintaxe $(( )) .
- Pode-se usar explicitamente o comando *test* em vez de [ ] .