ISCTE-IUL – Instituto Universitário de Lisboa
DCTI
Sistemas Operativos

Shell

JRG, fmmb, versão 1.04 – 12-04-2012

---

1. Linha de comando

Esta secção foca alguns aspectos de utilização corrente da linha de comando.

1.1 - Expansão. "Wildcards"

Um dos mecanismos mais usados da shell é a possibilidade de "apanhar" nomes de ficheiros usando o "`*`" ou, mais raramente, o "`?`".

De forma simples, o significado do "`*`" é "qualquer coisa". Mais exactamente, ao ler um nome com `*` a interpretação é: no lugar o `*` pode figurar qualquer sequência de caracteres, seja vazio ("nada"), um caractere qualquer, dois, etc.

A utilização mais vulgar do `*` é no `ls`. Por exemplo,

```
ls -l *.c
```

lista os nomes de ficheiro cuja nome adira ao padrão "qualquer coisa seguido de `.c`" (ou seja, os ficheiros com extensão `.c`). Mas pode ser usado em muitas outras circunstâncias como forma de apanhar um grupo de ficheiros. Por exemplo o comando

```
zip programas.zip *.c
```

cria um ficheiro zip com todos os `.c` existentes na directoria corrente.

1.2 - Escapes

O `*` e o `?` são caracteres especiais para a shell. Como estes há outros, por exemplo `$`, `&` ou >. Se tivermos a ideie não muito feliz, de dar nomes a ficheiros contendo caracteres deste tipo podemos ter alguns problemas.

Por exemplo, supondo que tínhamos a ideia não muito feliz de chamar a um ficheiro

```
***star***.c
```

a utilização simples do nome do ficheiro em comandos, por exemplo

```
vi ***star***.c
gcc ***star***.c -o ***star***
```

seria ambígua. Nesta situação teríamos que enquadrar o nome do ficheiro entre ' ':

```
vi '***star***.c'
gcc '***star***.c' -o '***star***'
```

O mesmo aliás se usarmos nomes de ficheiros com espaços. Por exemplo se tivermos a ideia também não muito boa de chamar a um ficheiro

```
exemplo 1.c
```

a utilização simples do nome do ficheiro, por exemplo:

```
vi exemplo 1.c
```

é também ambígua e resolve-se com

```
vi 'exemplo 1.c'
```

1.3 - Redireccionamento

A maioria dos comandos mandam o resultado para o ecrã. Por exemplo, ao fazer:

```
ls -l
```

o resultado do comando, a lista ficheiros existentes no directório corrente, vai para o ecrã. Se fizermos a seguinte variante

```
ls -l > lista.txt
```

nada aparece no ecrã; o resultado do comando vai, sim, para o ficheiro `teste.txt`. Dizemos que o resultado do comando é redireccionado para o ficheiro `lista.txt`.

Este é um exemplo de um mecanismo geral que a shell implementa e que permite redireccionar os canais "standard" de entrada e saída de um programa.

Normalmente a saída de um programa vai para o canal "standard de saída" (stdout) ou, em caso de erro, para o canal "standard de erros" (stderr). Um e outro podem ser redireccionados separadamente.

A forma básica deste mecanismo é o normal redireccionamento da saída. Ex:

```
cat /etc/passwd > x     #saída para o ficheiro x
cat /etc/passwd >> x    #saída acrescenta-se ao ficheiro x
```

No exemplo seguinte, a saída normal é redireccionada para `/dev/null`; e a de erros continua no ecrã. Se x não existir, a mensagem de erro vai para o ecrã;

```
cat x 1> /dev/null      # ou apenas: cat x > /dev/null
```

No exemplo seguinte, a saída de erros é redireccionada para `/dev/null`. Se o ficheiro existir, aparece no ecrã; caso contrário, não aparece nada no ecrã;

```
cat x 2> /dev/null
```

O exemplo seguinte, redirecciona a saída standard para `/dev/null` e a saída de erros para o mesmo sítio;

```
cat x > /dev/null 2>&1
```

Resta acrescentar que `/dev/null` é um ficheiro muito especial: representa um "poço sem fundo" (ou buraco negro) para onde se pode redireccionar qualquer output, que é descartado pelo Unix (e Linux);

Existe também o canal "standard de entrada" (`stdin`) que normalmente está associado ao teclado.

Nos exemplos seguintes, 1) é criado um ficheiro `x`; 2) o ficheiro `x` é usado como canal de entrada para o comando `cat`; 3) copia-se o conteúdo do ficheiro `x`, para um novo ficheiro `y`.

```
echo "teste" > x
cat < x
cat < x > y     ou     cat > y < x
```

Finalmente o | (designado "`pipe`") permite encadear comandos fazendo com que o standard output de um comando seja encaminhado para input do programa seguinte.
O exemplo seguinte mostra o encadeamento de 3 comandos: o primeiro mostra o conteúdo do ficheiro `/etc/passwd`; o segundo filtra esse conteúdo mostrando apenas as linhas que contêm a palavra `root`; o terceiro substitui ocorrências da palavra "`root`" pela palavra "`raiz`".

```
cat /etc/passwd | grep "root" | sed 's/root/raiz/'
```

1.4 - Variáveis de ambiente

Parte das variáveis de ambiente suportam a configuração de alguns aspectos do funcionamento da própria shell. Por exemplo a variável PS1 configura a prompt que a shell coloca no ecrã para pedir comandos.
Exemplo: o comando

```
PS1="diga>  "
```

altera a prompt para uma coisa do género:

```
diga >
```

Uma das variáveis mais importantes da shell é a `PATH`. Esta variável contém uma lista de nomes de directórios (separados por :). Quando é dado um comando a bash procura esse comando em cada um dos directórios dessa lista.

Mais especificamente, há dois tipos de comandos, internos e externos. Internos são os que a própria bash executa; por exemplo `cd`, `pwd` ou `exit` são comandos internos.

Quando damos um comando externo, estamos na prática, a pedir à bash para executar um outro programa. A bash responde a este pedido em dois passos:

1. tenta localizar o programa; não conseguindo dá um erro;
2. conseguindo, faz um `fork()` executando no processo filho o programa pedido;

Por exemplo, ao dar o comando `ls` estamos a pedir à bash para executar um programa chamado `ls` (sem dizer em que directório esse programa se encontra). A bash vai então procurar um ficheiro chamado `ls` em cada um dos directórios listados na variável `PATH`.
Por exemplo, imagine que a variável `PATH` tem este conteúdo:

```
/home/programas:/etc/aquinaoesta:/bin:/home/maisprogramas
```

ao dar o comando `ls` a bash vai procurar:

`/home/programas/ls`	não encontra
`/etc/aquinaoesta/ls`	não encontra
`/bin/ls`	encontra ! executa este programa (ou pelo menos tenta)
`/home/maisprogramas/ls`	aqui nem chega a procurar porque encontrou antes

É claro que a `PATH` só é relevante se o comando for dado com um nome simples. Se ao dar o comando indicar o nome do ficheiro explícita a `bash` não precisa procurar. Por exemplo, se der um comando:

```
/bin/ls
```

a bash tenta executar o programa que se encontre no ficheiro indicado.

Acontece também que, por defeito, o directório corrente não está incluído na lista de directórios da PATH. Isto significa que pode ter um programa mesmo ali à mão no directório corrente e a `bash` não o encontra - pela simples razão de que não o procura lá.

Exemplo:

`cc teste.c -o teste`	compila um programa, criando um programa teste
`teste`	a bash não localiza o programa no directório corrente
`teste: not found`	

Há várias formas de resolver isto. Uma é indicar o nome do programa explicitamente.

`./programa`	explicitar a localização do ficheiro (comando) a executar

Outra forma é resolver o problema de vez, adicionando o directório corrente à lista de directórios onde a `bash` procura comandos:

`PATH=$PATH:.`	juntar mais um elemento (o . ponto) à lista de directórios

1.5 - subshell; exit

A shell é o programa interactivo que aceita e promove a execução dos nossos comandos. Há vários programas diferentes para este efeito (várias shell). A que vamos utilizar é a **bash**. Outras hipóteses são **sh**, **csh**, **tcsh**, **zsh**. Todos estes programas se encontram, normalmente, no directório `/bin`.
Exemplo: experimente o comando

```
ls -l /bin/*sh*
```

Normalmente a shell é lançada, automaticamente, quando fazemos o *login* num terminal ou quando abrimos uma nova janela de comandos num ambiente gráfico; e termina quando damos o comando `exit`.

Exemplo: experimente o comando `ps`, para ver os processos em curso no terminal/janela de comandos. Um deles será a shell (muito provavelmente o único além do próprio comando `ps` em curso no momento).

Podemos, entretanto, lançar novos processos shell executando o comando respectivo.
Exemplo: experimente e interprete a seguinte sequência

`/bin/bash`	lançar uma nova shell
`ps`	deverá haver pelo menos dois processos shell em curso
`exit`	
`exit`	oops...

2. Programação

Mecanismos de programação

2.1 - scripts

Os comandos para a shell, que normalmente são executados interactivamente, podem também ser mandados executar através de um ficheiro.

Exemplo: escreva o seguinte conjunto de comandos num ficheiro `teste1`

```
#!/bin/bash
echo -n "Data: "
date "+%d de %B de %Y"
echo "LISTA DE UTILIZADORES"
who
```

Para executar estes comandos podemos agora mandar "executar o ficheiro".

Normalmente será necessário, primeiro, atribuir a permissão x ao ficheiro. Trata-se de um ficheiro de texto e por isso, à partida, não é normal que tenha a permissão x. Podemos atribuí-la com

```
chmod +x teste1
```

ou, por exemplo,

```
chmod 755 teste1
```

Feito isso podemos então mandar executar o ficheiro com

```
./teste1
```

ou simplesmente

```
teste1
```

se o directório corrente estiver na variável `$PATH`.

O efeito será idêntico ao que seria obtido pela execução dos comandos, um por um, na linha de comando.

2.2 - Variáveis

2.2.1 - Criação

Um script simples pode ser uma lista de comandos. Na realidade a shell inclui uma série de outros mecanismos que permitem configurar uma espécie de linguagem de programação.

O primeiro desses mecanismos de programação é a possibilidade de criação de variáveis.

Para definir uma nova variável utiliza-se um comando com a sintaxe:

```
nome_da_variavel=valor
```

Exemplo: Experimente a seguinte sequência de comandos:

```
x=Hello
set
```

O primeiro cria uma variável da shell com nome `x` e conteúdo `Hello`; o segundo comando mostra a lista de variável, onde já deve aparece a recém-criada variável `x`.

Para ver o valor de uma variável pode-se usar o comando `echo`. Na sua forma mais simples este comando, `echo`, permite escrever um texto para o ecrã. Por exemplo:

```
echo Hello World
```

escreve no ecrã, o texto `Hello World`.

No texto a escrever pode-se incluir uma variável da shell. Para mencionar a variável escreve-se o nome antecedido de $. Por exemplo:

```
echo $USER
echo Eu sou, portando, o utilizador $USER
echo Eu sou o utilizador  $USER e estou a  executar a shell $SHELL
```

Exemplo: experimente e interprete a seguinte sequência de comandos:

```
x=ABC
echo $x
echo x=$x
echo x dá x e \$x dá $x
x=123
echo x foi modificado para $x
```

2.2.2 - Manipulação de variáveis

As variáveis podem-se alterar (podem-se redefinir com outro valor). Por exemplo, experimente e interprete a seguinte sequência:

```
x=123
echo $x
x=456$x
echo $x
x=$x456
echo $x
```

O valor a atribuir a uma variável pode ser colocado entre " ou ' (que, como habitualmente não ficarão a fazer parte do conteúdo). Este mecanismo é útil para incluir na variável caracteres que possam ter significado especial. Exemplo: o comando

```
x=    ABC
```

não funciona para incluir espaços na variável e aliás dá erro. Pode fazer:

```
x="   ABC"
```

As variáveis da shell representam sequências de texto simples. Em geral o conteúdo de uma variável não é interpretado pela shell (ou seja, é usado literalmente).
Exemplo - experimente e interprete a seguinte sequência:

```
x=1
y=2
z=$x+$y
echo $z
```

Ocasionalmente, pode-se gerar ambiguidade na identificação do nome de uma variável. Nesse caso pode-se enquadrar no nome entre os caracteres { }.
Exemplo, experimente e interprete a seguinte sequência:

```
x=abc
y=$xdef
echo $y
y=${x}def
echo $y
```

Uma variável pode ser destruída com o comando `unset`. A utilização de uma variável não definida não provoca qualquer erro - origina, simplesmente, uma cadeia de texto vazia.
Exemplo - experimente e interprete a seguinte sequência:

```
x=ABC
set                        x deve aparecer na lista de variáveis
echo Valor de x = $x
unset x
set                        x já não deve aparecer na lista de variáveis
echo Valor de x = $x
```

2.2.3 - Operações aritméticas

Muitas vezes é necessário fazer contas nos scripts em shell, tal como em qualquer linguagem de programação. O comando `$(( ... ))` permite obter o resultado de expressões numéricas;
Exemplo:

```
x=5
echo $(( $x+1 ))
```

2.2.4 - Variáveis de ambiente

Identicamente, o script herda as variáveis de ambiente do processo original. Estas variáveis podem ser usadas e alteradas, apenas com efeitos durante a execução.

Considere a seguinte sequência:

```
export x=1
y=2
bsh_1c
echo "x=$x; y=$y; PATH=$PATH"
```

Elabore um script `bsh_1c` que mostre que:
- Pode usar e alterar as variáveis x e PATH. E a variável y ?
- Estas alterações não têm efeito no processo original;

As variáveis da shell podem-se confinar a um processo ou ser transmitidas aos processos descendentes. A estas últimas chamamos variáveis de ambiente.
Para criar uma variável de ambiente utiliza-se o comando export, com a sintaxe:

```
export nome-da-variavel=valor
```

Por exemplo, o comando

```
export v="Teste"
```

Cria uma variável de ambiente `v`.

Exemplo - experimente e interprete a seguinte sequência

```
x=ABC
export y=ABC
set                 x e y figuram na lista de variáveis da shell em que foram criadas
/bin/bash           abrir uma nova shell
set                 apenas y figura na nova shell (processo filho) entretanto criada
exit                voltando à shell original...
set
```

As variáveis de ambiente são transmitidas aos processos descendentes por cópia. O processo pode modificar a variável recebida, mas essa modificação não se reflecte na variável existente no processo original.
Exemplo - experimente e interprete a seguinte sequência

```
export y=ABC
/bin/bash           abrir uma nova shell
echo $y
y=123
echo $y
exit                voltando à shell original
echo $y
```

2.3 - Argumentos

Um dos mecanismos fundamentais para a construção de scripts são os argumentos passados na linha de comando. Estes argumentos são recebidos no script em variáveis especiais, designadas por:

```
$1   $2   $3    $4 ...
```

exemplo: construa o seguinte script (bsh_1d)

```
#!/bin/bash
echo "Primeiro argumento: $1"
echo "Segundo argumento: $2"
echo "Terceiro argumento: $3"
echo "Sétimo argumento: $7"
echo "Décimo segundo argumento: ${12}"
```

Experimente com:

```
bsh_1d a b c
bsh_1d 1 2 3 4 5 6 7 8 9 10 11 12 13
```

↳ Construa um script para verificar o significado das variáveis especiais $0, $# e $*

A construção dos argumentos é feita após a substituição dos símbolos especiais pela shell;
Verifique o valor dos argumentos nas seguintes situações:

```
bsh_1d ~
bsh_1d *
bsh_1d  O meu nome de utilizador é $USER
bsh_1d  "O meu nome de utilizador é $USER"
bsh_1d  $USER $NADA             # $NADA não está definido
bsh_1d  '$USER' $USER
bsh_1d  '$USER $NADA $HOME'
bsh_1d  "$USER $NADA $HOME"
bsh_1d  O $NADA NAO EXISTE
bsh_1d  O "$NADA" PODE EXISTIR
```

2.4 - read

Outro dos mecanismos importantes para construir um script é o comando `read`, que permite ler interactivamente um valor para uma variável (uma espécie de `gets` da BASH);

```
#!/bin/bash
echo "Diga qualquer coisa: "
read x
echo "Disse: $x"
```

2.5 - Estruturas de controlo

2.5.1 - if

Além de alinha comandos em sequência, a shell permite a utilização dos mecanismos de controlo habituais numa linguagem de programação, designadamente if´s e ciclos.

Na sua forma básica o if permite escolher, para execução, um de dois grupos de comandos alternativos. A forma geral é:

```
if comando
then
      lista-comandos-1
else
      lista-comandos-2
fi
```

Exemplo:

```
if pwd
then
      echo "o pwd funciona"
else
      echo "dificilmente alguém verá isto"
fi
```

↳faça um script bsh_2d que:
- recebe um argumento
- faz chmod 700 do argumento
- se correu bem diz: "Comando executado com sucesso";
- caso contrário diz: "Comando falhou";

O comando `test` é especialmente vocacionado para a construção de condições de `if`. Trata-se de um comando que, na prática, serve para verificar uma condição produzindo um resultado adequado á utilização na estrutura do `if`;

Exemplo:

```
echo -n "Username: "
read $x
if  test $x = $USER                  # ou if [ $x = $USER ]
then
  echo "Acertou."
else
  echo "Falhou."
fi
```

Esta forma do comando `test` permite verificar se duas "strings" são iguais. Se forem, o resultado da execução do comando será "sucesso" (representado por 0). Caso contrário, será "insucesso" (representado pelo valor 1);

```
test "abc" = "xyz"      #ou [ "abc" = "xyz" ]
echo $?
[ "abc" = "xyz" ]       # ou test "abc" = "xyz"
echo $?
[ "abc" != "xyz" ]      # ou test "abc" != "xyz"
echo $?
```

O comando tem duas sintaxes alternativas: utilizar o comando test ou colocar os argumentos entre [ ]; a partir daqui vamos sempre usar esta última; mas não se esqueça que [...] representa um comando test !

As operações -z e -n permitem verificar, respectivamente, se uma string é vazia (tamanho 0) ou é não vazia (tamanho > 0)

Exemplo:

```
#!/bin/bash
echo -n "Nome: "; read x
if [ -z "$x" ]; then
echo "(vazio)"
else
if [ $x -eq $USER] ; then
echo "Acertou."
else
echo "Falhou."
fi
fi
```

O exemplo anterior usa dois ifs encadeados.
Em alternativa pode ser usada a sintaxe `if-elif-else-fi` ilustrada no seguinte exemplo:

```
#!/bin/bash
echo -n "Nome: "; read x
if [ -z "$x" ]; then
      echo "(vazio)"
elif  [ $x = $USER]; then
      echo "Acertou"
else
       echo "Falhou."
fi
```

O mesmo comando permite comparar números inteiros; algumas das opções são exemplificadas no script seguinte:

```
#/bin/bash
x=700
echo -n "diga um número: "; read n
if [ $n -lt $x ] ; then               # -lt : less then
   echo "Abaixo"
elif [ $n -gt $x ] ; then             # -gt : greater then
   echo "Acima"
else
   echo "Certo."
fi
```

(para uma lista completa dos operadores veja o manual do comando `test`)

↳ Faça outras versões do mesmo script usando as opções `-eq` e `-gt`;

↳ Faça um script que receba dois argumentos, ambos números inteiros, e escreva os mesmos dois números por ordem (e apenas um deles, se forem iguais);

2.5.2 - Teste com ficheiros

Um outro grupo importante de opções do comando test serve para testar condições sobre ficheiros; Por exemplo, o seguinte script verifica se um dado ficheiro existe:

```
#!/bin/bash
if [ -f $1 ]; then
      echo "$1 existe"
else
      echo "$1 não existe"
fi
```

O seguinte script exemplifica algumas das opções mais do test para ficheiros;

```
if [ -z $1 ] ; then     # falta se não houver argumento
   echo "usage: $0 ficheiro"
   exit  # terminar o script
fi

if [ -f $1 ] ; then; echo "$1 é um ficheiro normal"; fi
if [ -d $1 ] ; then; echo "$1 é um directório"; fi
if [ -r $1 ] ; then; echo "$1 term permissão r"; fi
if [ -w $1 ] ; then; echo "$1 tem permissão w"; fi
if [ -x $1 ] ; then; echo "$1 tem permissão x"; fi
```

2.5.3 - Case

A instrução case permite escolher um de vários blocos de comandos alternativos, o sua forma geral é a seguinte:

```
case $variavel in
caso1)
  lista-comandos1;;
caso2)
  lista-comandos2;;
...
esac
```

O seguinte exemplo ilustra a utilização do case:

```
#!/bin/bash
case $1 in
      benfica) echo "Lisboa";;
      sporting) echo "Lisboa";;
      porto) echo "Porto";;
      boavista) echo "Porto";;
      esac
```

No exemplo anterior as opções são valores unívocos. De uma forma geral, as opções podem ser um padrão; nesse caso, é seleccionada a primeira opção que adira à string de controlo (indicada na linha `case...in`).

A construção do padrão admite mecanismos que são mais ou menos familiares:

| - alternativas (or)
* - qualquer cadeia de caractares (0 ou mais)
? - um caracter qualquer

```
#!/bin/bash
case $1 in
      benfica|sporting) echo "Lisboa";;
      porto|boavista) echo "Porto";;
      *) echo "Outras cidades";;
esac
```

O padrão * captura todas as sequências, conseguindo-se assim uma forma de obter um "default";

↳ O que acontece se trocar a ordem das opções pondo o *) em primeiro lugar ?

Um outro exemplo... o que faz ?

```
#!/bin/bash
case $1 in
      *.txt) echo "ficheiro de texto";;
      *.C|*.c|*.h) echo "c/c++";;
      *) echo "outros ficheiros";;

esac
```

2.5.4 - For

O comando for permite repetir um conjunto de comandos, para cada um dos elementos de uma lista. A sua sintaxe é

for variavel in lista
do
    lista-de-comandos
done

Exemplo:

```
for i in "a b c"
do
  echo "i= $i"
done
```

Muitas vezes o for é utilizado para percorrer a lista de ficheiros de um directório. Para formação dessa lista aplicam-se os mecanismos habituais de expansão da shell:

```
#!/bin/bash
for i in *
do
      echo "entrada: $i"
done
```

Exemplo: listar ficheiros (excluindo os directórios)

```
#!/bin/bash
for i in *; do
  if [ ! -d $i ] ; then
     ls -l $i
  fi
done
```

↳ altere para listar apenas os ficheiros de extensão .c, .C ou .h

Como vimos anteriormente, as variáveis especiais $1, $2, ... representam os argumentos do comando. Estas variáveis são adequadas para aceder aos argumentos um a um. Para aceder aos argumentos num ciclo são úteis as seguintes outras variáveis especiais:

$* - todos os argumentos
$# - o número de argumentos

```
#!/bin/bash
echo "foram dados $# argumentos, que são: "
for a in $* ; do
  echo "-> $a"
done
```

2.5.5 - while

O ciclo `while` é um ciclo de condição: repete a execução de um bloco de comandos enquanto se verificar uma dada condição de controlo (ou até ela deixar de se verificar);

while [condição]
do
    lista-de-comandos
done

Exemplo:

```
#!/bin/bash
y=700
x=0
while [ $x != 700 ]
do
  echo -n "Adivinhe o numero: "; read x
done
```

Altere o exemplo anterior para fazer um pequeno jogo de aproximação: após cada tentativa o script deve ajudar dizendo se o valor certo é para "Cima" ou para "Baixo".

2.5.6 - Substituição

Os delimitadores ` ` substituem um comando (escrito entre os dois delimitadores) pelo seu output; exemplo

```
x=`date`
echo $x
```

Este mecanismo é um precioso auxiliar para scripts. Por exemplo, pode ser utilizado para o backup de um ficheiro com a data actual.

3. Comandos

Alguns comandos de apoio a scripts

3.1 - expr

O comando `expr` (`/bin/expr`) contempla também algumas operações com strings; em particular é a forma mais fácil de fazer uma operação importante que é localizar uma string dentro de outra;

Exemplo: o script seguinte lê uma string e indica em que posição se encontra a letra "a":

```
#!/bin/bash
echo -n "String: "
read s
echo `expr index $s a`
```

Na realidade o comando `expr` serve para fazer o cálculo de expressões simples, quer em texto quer em números inteiros, sendo portanto, também, um alternativa ou complemento ao cálculo numérico com `$(( ))`;

Ex: o seguinte script lê uma string, representando o nome completo de uma pessoa, e mostra apenas o primeiro nome próprio:

```
#!/bin/bash
echo  -n "String: "
read s
n=`expr index "$s" " "`
if [ $n -gt 0 ] ; then
    m=`expr $n - 1`
fi
```

exercício (faça todos os cálculos com `expr`);
- altere para mostrar o primeiro e último nome;
- altere para mostrar o último apelido e depois todos os outros nomes;

3.2 - grep

O comando `grep` permite procurar uma string num ficheiro. Na forma normal o comando mostra as linhas do ficheiro que contêm esse padrão. Com a opção -c mostra o número de linhas nas mesmas condições;

Exemplo:

```
grep $USER /etc/passwd
grep -c $USER /etc/passwd
```

↳ Faça um script que indique o número de ficheiros .c , .h existentes no directório;

↳ Faça um script que receba um argumento e indique cada um dos ficheiros .c onde o argumento aparece e quantas vezes;

↳ Faça um script que indique os ficheiros .h pendurados (não incluídos em qualquer ficheiro .c ou .h);

↳ Faça um script que indique, para cada ficheiro .h, a lista dos ficheiros (.h ou .c) onde é incluído;

3.3 - basename

Há um conjunto de comandos que, embora possam ser usados interactivamente, são especialmente importantes no quadro da programação com a shell; é o caso, por exemplo, do comando `basename`;

o comando extrai o nome base de um nome completo de ficheiro, retirando:
- o caminho até ao directório;
- a parte final (se for igual ao segundo argumento)

```
basename  /etc/passwd           #passwd
basename  teste.c  .c           #teste
basename  /etc/rc.1 .1          #rc
basename teste.c .x       #teste.c
```

3.4 - wc

O comando `wc` dá indicações sobre a dimensão de um ficheiro: tipicamente o número de linhas, palavras e caracteres; a opção -l dá apenas o número de linhas;

exemplo:
```
cat /etc/passwd | wc
cat /etc/passwd | wc -l
```

↳ faça um script que indique o número de linhas de código do projecto (número total de linhas nos ficheiros .h e .c);

3.5 - sed

Alguns comandos do Unix são especialmente importantes quando se trata de scripts; Um desses comandos é o `sed`, que introduz também a noção de expressões regulares;

Essencialmente, o `sed` lê uma entrada e produz uma saída transformada;
A transformação a fazer é indicada num comando dado como argumento `sed`;
Há duas transformações importantes:
- Seleccionar parte das linhas do ficheiro original;
- Substituir parte do texto do ficheiro original;

o seguinte comando selecciona as linhas do ficheiro `/etc/passwd` que contêm a palavra `root`;

```
sed  -n  '/root/p'  /etc/passwd
```

A opção `-n` faz com que o `sed` reproduza apenas as linhas em que isso é pedido explicitamente; neste caso aparecem apenas as linhas seleccionadas pelo comando que tem a forma:
```
/expressão-regular/p
```

o efeito deste comando é escrever (o "p" é de print) as linhas que aderem à expressão-regular indicada; neste caso aderem à expressão regular as linhas contendo a palavra indicada;

Numa expressão-regular podem-se usar símbolos com significado especiais que permitem bastante flexibilidade na localização de sequências de texto; essa simbologia resume-se na seguinte tabela:
- * Qualquer sequência de caracteres;
- ? - Um caracter qualquer
- ^ - Início da linha
- $ - Fim da linha
- [ ] - um dos caracteres indicados;
- \x - escape de um caractere especial; ex \$ para representar o caractere $

Os [ ] denotam um caractere, dentro dos indicados na lista; Por exemplo [abc] denota um caractere, que seja a ou b ou c;
Na lista podem figurar intervalos; ex: [a-z] denota uma letra minúscula e [a-zA-Z] uma letra qualquer;
O símbolo ^ antes da lista denota todos os caracteres menos os indicados;
ex: [^abc] qualquer caractere que não seja o a ou b ou c;
ex: [^a-b] qualquer caractere que não seja uma letra minúscula;

Exemplos:

^a	localiza as linhas começadas por a (ie: contendo a sequência "início da linha a");
a$	as linhas terminada em a (ie: contendo a sequência "a fim de linha");
[abc]	um a ou um b ou um c;
^[0-9]	Começadas por algarismo;
^?$	contendo apenas um caracter;
^[a-zA-Z]$	linha contendo apenas uma letra;
Maria	a palavra Maria
[Mm]aria	a palavra Maria ou a palavra maria

Exemplos
Listar apenas as directorias

```
ls -l  |  sed -n '/^d/p'
```

Listar os ficheiros executáveis

```
ls -l | sed  -n  '/^??x??x??x/p'
```

O comando seguinte exemplifica a utilização do `sed` para transformar o conteúdo de um ficheiro:

```
sed 's/a/Aluno/' /etc/passwd
```

Neste caso o comando para o `sed` é `s/a/Aluno` e tem como efeito substituir a por Aluno;
(como é evidente, o `sed` não altera o ficheiro; apenas o lê e mostra o seu conteúdo transformado);

Este comando sed tem a forma geral:

```
s/expressão-1/expressão-2/
```

e tem como efeito substituir a expressão-1 do texto original pela expressão-2;
(`s/expressão-1/expressão-2/g` para substituir todas as ocorrências de expressão-1 na mesma linha);

Exemplo: eliminar a palavra “do” e substituir todos os “a” por “AA”

```
echo “maria do carmo joaquina” | sed ‘s/do //g;  s/a/A/g’
```

3.6 - awk

O awk é um comando muito amplo que engloba, ele próprio, uma linguagem para programação de scripts de manipulação de texto; vamos apenas algumas construções simples/típicas com awk;

O exemplo seguinte mostra apenas uma parte (2 das colunas) do resultado do comando ls -l;

```
ls -l | awk ' {print $1 $3 ; }'
```

a exemplo do sed, também o awk aceita um "comando" passado como argumento;
o comando é indicado entre { } e contém uma ou mais instruções, que terminam com ;
Neste caso a instrução é o print que "imprime" no ecrã;

o que importa aqui é o facto de o awk, ao processar cada linha, dividir os campos (separados por espaços) em variáveis $1 $2 etc; assim, ao escrever estas variáveis, estamos a escrever as colunas 1 e 3 do resultado do ls -l;
O exemplo anterior imprime todas as linhas; uma variante é um comando da forma:

```
/expressão-regular/ { comandos }
```

que aplica o bloco de comandos apenas à linhas seleccionadas pela expressão-regular. Exemplo:

```
> ls -l  | awk '/^d/ { print $1 $3 }'
```

Mostra o mesmo conteúdo mas apenas para os directórios;

É possível associar um bloco de comandos a diferentes condições de selecção. Por exemplo:

```
ls -l  | awk '
  /^d/ { print $1 $3; }
  /^-/  { print ficheiro $2;  }'
```

Trata de forma diferente directórios e ficheiros comuns;

É ainda possível associar um bloco de comandos a uma situação especial, denotada pela marca BEGIN que ocorre antes da leitura de entrada; isto é útil para fixar alguns parâmetros importantes como o FS que indica o caracter de separação dos campos:

Exemplo:

```
cat /etc/passwd | awk 'BEGIN { FS=":" ; } { print $1 $5; }'
```

Mostra o nome e descrição de cada utilizador (colunas 1 e 5 do ficheiro /etc/passwd);

Exemplo:

Considere o ficheiro users.txt com o seguinte conteúdo.

```
josesilva:sporting:1212-1212-121212
cebola:x:2333-2222-2323223
tintim:stromp:2323-2323-2323232
jaquim:ola:2232-1111-11111111
pedro:benfica:3333-3333-3333333
```

O comando seguinte extrai apenas a coluna 2:

```
cat users.txt| awk -F':' '{print $2}'
```

4. Outros mecanismos. Complementos.

4.1 - Expansão

Questão muito importante: quando fazemos um comando como o ls -l *.c ou zip programas.c *.c quem é que transforma o *.c numa lista de nomes de ficheiros, a shell ou os comandos. Resposta: a shell. Por exemplo, ao fazermos

```
ls -l *.c
```

a shell vai invocar o comando ls passando-lhe como argumentos todos os nomes de ficheiro que encontrar que adiram ao padrão *.c (ou seja o nome de todos os ficheiros com extensão .c que encontrar, neste caos no directório corrente).

Podemos. em caso de dúvida, fazer um programa nosso para o demonstrar. O seguinte programa escreve no ecrã os argumentos que receber da linha de comando:

```
// eco
#include <stdio.h>
int main( int argn, char *argv[]) {
    int i;
    for ( i = 1; i < argn; i++ ) {
        printf ("%d: %s\n", i, argv[i] );
    }
}
```

Experimente, por exemplo:

```
./eco a b c
./eco *.c
```

MAIS AVANÇADO

Muitas vezes é preciso impedir a shell de seguir o procedimento normal de expansão de caracteres. Por exemplo, para escrever um `$` no ecrã é preciso indicar à shell para não interpretar o `$` como iniciador do nome de uma variável. Diz-se então que estamos a fazer o escape (do significado habitual) do caracter.
Uma das maneiras de escapar uma sequência é inseri-la entre `' '`. Por exemplo:

```
x=bifa
echo $x     # o  $X é expandido
echo '$x'   # os '  ' impedem a expansão
```

O mecanismo de expansão deriva da presença de caracteres com significado especial para a shell. No caso anterior o significado especial é dado pelo $. As **''** retiram o significado especial ao $ e por isso $x fica a ser só, apenas e literalmente $x.

(Em rigor, as `' '` são necessárias apenas para lidar com o `$`. Ou seja, poderíamos obter o efeito desejado apenas com `echo '$'x`. Mas `'$x'` também funciona e é muito mais claro.)

(São também caracteres especiais da shell os seguintes:

```
*  ?  [  ]  '  "  \  $  ;  &  (  )  |  ^ <  >  {  }
```

que normalmente, para serem assumidos de forma literal, devem também ser *escapados*).
Há 3 elementos sintácticos que impedem a expansão: as `" "`, as `' '` e a `\`.

As aspas são as mais fracas: escapam apenas alguns caracteres. Não escapam, por exemplo, o `$` e, por isso, as variáveis colocadas entre aspas continuam a ser expandidas.
Por exemplo:

```
echo "*** $HOME ***"
```

torna os `*` literais, mas mantém a expansão da variável `$HOME`

As `' '` escapam todos os caracteres e o `\` escapa o caracter seguinte (apenas 1) seja ele qual for.
Por exemplo:

```
echo ' $HOME'
echo \$HOME
```

(para escrever \ usa-se \\).

4.2 - *if / test / review*

É importante perceber a subtileza da "condição" do if. Numa linguagem de programação clássica a decisão do if é baseada numa condição. Na shell a decisão é baseada no resultado da execução de um comando.

Exemplo: considere o seguinte comando

```
> chmod 700 x
```

Se o ficheiro x existir (e houver permissão) o comando é executado com sucesso; caso contrário, a operação não é executada e aparece uma mensagem de erro. Verifique o efeito no script seguinte (`bsh_2b`)

```
#!/bin/bash
echo "Ficheiro: "
read x
if chmod 700 $x
then
  echo "Comando executado!"
else
  echo "Como deve ter percebido, não correu bem."
fi
```

A variável especial **$?** representa a o resultado do último comando executado. Veja o valor de $? após o comando

```
chmod 700 x
echo $?
```

(veja para o caso de sucesso e para o caso de insucesso na execução do comando).

↳ qual o problema com este script ?

```
#!/bin/bash
chmod 700 x
if $?
then
  echo "Ok"
else
  echo "Erro"
fi
```

4.3 - exit; encadeamento de comandos;

Interactivamente o comando **exit** serve para terminar uma shell; ex:

```
/bin/bash   # inicia nova shell
exit        # termina (volta à shell inicial)
```

num script tem o efeito correspondente, ie, termina a execução do próprio script; ex:

```
#!/bin/bash
date
exit              # o script termina aqui
echo "Bye."       # este comando não chega a ser executado
```

O exit pode ser utilizado para terminar um script, prematuramente, em condições anormais.

Exemplo: o seguinte script compila um programa em C, criando um executável com o mesmo nome; o nome do ficheiro é dado num argumento para o script; o primeiro passo é justamente verificar o argumento:

```
#!/bin/bash
if [ ! $#  -eq 1 ] ; then
      echo "usage: $0 <file>"
      exit
fi
cc  $1 -o   `basename $1 .c`
```

Além da função de instrução de controlo, o `exit` tem outro efeito importante que é estabelecer o resultado – o *`exit` status* do comando;

Exemplo: considere o seguinte script com o nome *trivial*

```
#!/bin/bash
exit $1
```

verifique o efeito da seguinte sequência:

```
trivial 2
echo $?           # dá ?
if [ trivial 0 ] ; then ; echo "funciona."; fi
```

O símbolo **$?** representa o *`exit` status* do último comando a ser executado

Considere a seguinte versão do exemplo b):

```
#!/bin/bash
if [ ! $#  -eq 1 ] ; then
  echo "usage: $0 <file>"
  exit 1
fi
cc $1 -o `basename $1 .c`
exit 0
```

- se não for dado um argumento o script termina com saída em de erro (diferente de 0);

- ao contrário, se tudo correr bem, termina com 0; note que o exit 0 final não tem interesse de execução (não adianta fazer exit quando o script vai terminar "sozinho"); o interesse é apenas estabelecer o resultado de saída em situação normal - ou seja, 0;
- em vez de exit 0 não seria melhor exit $? ?

Os elementos && e || permitem fazer o encadeamento de comandos numa lógica parecida com a dos operadores homólogos da linguagem C;

A sequência:

```
comando1 && comando2 && ...
```

executa os comandos por ordem enquanto derem "sucessso" (ou seja, termina a sequência se um deles falhar)

A sequência:

```
comando1 ||  comando2  ||  ....
```

executa os comandos por ordem até um deles ter sucesso (ou seja, executa os comandos por ordem enquanto falharem);

Exemplo:

```
trivial 0 && echo "Ok."
trivial 1 && echo "NOP"
trivial 0 ||  echo "NOP"
trivial 1 ||  echo "Ok."
```

A seguinte versão do script *comp* verifica se é dado um argumento e se o ficheiro correspondente existe; caso uma das condições não se verifique o script termina com erro:

```
#!/bin/bash
if [ ! $# -eq 1 ] || [ ! -f $1 ] ; then
            echo "$0: invalid arguments"
            exit 1
fi
cc $1 –o `basename $1 .c`
exit $?
```

Uma alternativa poderia ser:

```
#!/bin/bash
if [ $# -eq 1 ] &&  [ -f $1 ] ; then
            :
else
            echo "$0: invalid arguments"
            exit 1
fi
cc $1 –o `basename $1 .c`
exit $?
```

Verificadas as duas condições é executado o comando : (que "não faz nada") e o script segue depois do if; de contrário, termina com exit;

O comando

```
comp teste.c && teste
```

compile o programa e, em caso de sucesso, execute-o;

4.4 - Funções

As funções são um elemento estruturante parecido com as funções das linguagens de programação clássicas; no caso dos scripts, enquadram um conjunto de comandos que são executados quando a função é invocada através do seu nome; ex:

```
#!/bin/bash
exemplo() {
  echo $FUNCNAME says hello
}

echo "chamar a função..."
exemplo
echo "repete..."
exemplo
```

As funções aceitam argumentos numa sintaxe muito semelhante à dos próprios argumentos dos scripts; exemplo:

```
#!/bin/bash
say () {
  echo "I say,  " $1
}

say hello
say hello hello
say "helo hello hello"
```

O comando **return** termina a função, em determinado ponto, permitindo também formar um resultado de retorno; ex:

```
#!/bin/bash
say () {
  if [ $# -eq 0 ] ; then
    echo "nothing to say"
    return 1
  fi
  echo "I say,  " $*
  return 0
}

say hello
say hello hello
say "helo hello hello"
say
echo  just say `say`
```

As variáveis do script (as que são herdadas ou as que são criadas no próprio script) estão disponíveis na função; podem ser aí alteradas tal como podem ser criadas novas variáveis; as variáveis trabalhadas deste modo são todas "globais" (no sentido que o termo tem na programação clássica): existem podem ser criadas, alteradas e destruídas em todo o lado;

Exemplo:

```
#!/bin/bash
f () {
  echo "$FUNCNAME : Y= $Y"
  X=77
  Y=88
  echo "$FUNCNAME : X= $X"
```

```
  echo "$FUNCNAME : Y= $Y"
}

Y=66
echo "Y= $Y"
f
echo "X= $X"
echo "Y= $Y"
```

Exemplo: no seguinte script é feita uma função *readline* que lê uma string:

```
#!/bin/bash
readline () {
        echo "readline..."
        msg=""
        if [ $# -gt 0 ] ; then
           msg="$*: "
        fi

        str=""
        while [ -z $str ] ; do
           echo -n $msg
           read str
        done
        STR=str
}

readline "Teste"
echo "Lido: " $STR
```

4.5 - Arrays

Nos scripts podem-se usar variáveis indexadas; ex:

```
#!/bin/bash
a[0]="Hello"
a[3]="World"
echo "${a[0]} ${a[3]}"
```

note a utilização da sintaxe **${}** para isolar o nome das variáveis;

É raro haver interesse em usar variáveis indexadas, isoladamente, em vez de variáveis comuns. Normalmente o que se pretende é usar um conjunto de posições contíguas em processamentos iterativos (ou seja, o correspondente aos arrays nas linguagens de programação clássicas);

Exemplo: gerar 6 números aleatórios:

```
#!/bin/bash
i=0
while [ $i  -le 5 ]  ; do
      num[i]=$RANDOM
      echo "Número : $i ${num[i]}"
            i=$(( $i + 1))
done
```

A variável $RANDOM fornece um número aleatório ("diferente") de cada vez que é usada; o número gerado situa-se entre 0 e 2**15 (2^{15});

Um `while` deste género pode ser escrito de maneira mais familiar com a seguinte sintaxe alternativa, mais simpática para ciclos iterativos;

Exemplo: o seguinte script gera 6 números aleatórios entre 1 e 20:

```
#!/bin/bash
for (( i=0; i < 5; i++ )) ; do
  num[i]=$(( 1 + 20 * $RANDOM / 2**15 ))
done
```

Exemplo: o seguinte script gera 6 números aleatórios, entre 1 e 20, apresentando-os por ordem;

```
#!/bin/bash
for (( i=0; i < 5; i++ )) ; do
  num[i]=$(( 1 + 20 * $RANDOM / 2**15 ))
done

for (( i=0; i < 5; i++ )) ; do
  for (( j=0; j < 5; j++ )) ; do
    if [ ${num[i]} -lt ${num[j]} ] ; then
      x=${num[i]}
      num[i]=${num[j]}
      num[j]=$x
    fi
  done
done

for (( i=0; i < 5; i++ )) ; do
  echo "Numero: $i ${num[i]}"
done
```

Exercício: faça um script que gere uma aposta do totoloto, ie, 6 números diferentes, entre 1 e 49;

4.6 - Strings

length – obter o comprimento, ie o número de caracteres, de uma string;

O exemplo seguinte mostra o tamanho de uma string lida

```
#!/bin/bash
echo -n "String: "
read s
echo ${#s}
```

substring – extrair parte de uma string;

o exemplo seguinte lê uma string e mostra parte dos caracteres lidos:

```
#!/bin/bash
echo  -n "String: "
read s
echo ${s:5}
echo ${s:5:3}
n=${#s}
if [ $(( n % 2 )) -eq 1 ] ; then
  m=$(( n / 2 ))
  echo ${s:$m:1}
fi
```

Exercício: altere para mostrar também:
- o último caractere
- a primeira metade da string;

substituição – substituir uma parte da string

o exemplo seguinte substitui a letra a pela letra x na string lida;

```
#!/bin/bash
echo  -n "String: "
read s
s1=${s/a/x} ; echo $s1
s1=${s//a/y} ; echo $s1
```

experimente uma entrada com várias letras a para verificar a diferença entre / e //;

4.7 - Separação de palavras

Evidentemente que exercícios como o anterior são mais fáceis de realizar usando os mecanismos da shell que naturalmente separam palavras; ex:

```
#!/bin/bash
echo  -n "String: "
read s
p=""
for i in $s ; do
  if [ -z $p ] ; then
    p=$i; echo "Primeiro: $p"
  fi
done
echo "Ultimo: $i"
```

Estes métodos ganham ainda maior utilidade com a possibilidade de escolher o separador de palavras, que pode ser indicado à shell na variável IFS; normalmente o separador é o espaço, mas pode-se alterar através desta variável;

Exemplo: o seguinte script mostra a lista de directório da PATH, um por linha:

```
#!/bin/bash
IFS=:
for d in $PATH ; do
echo $d
done
```

4.8 - Exercícios

4.8.1 - Acertar na soma

Faça um script em bash que atribua números aleatórios às variáveis x e y, faça a sua soma e peça ao utilizador para adivinhar o resultado. Quando o utilizador acertar, o script deverá indicar o tempo que foi usado para fazer a conta em segundos.

```
#/bin/bash
x=$(( $RANDOM / 100 ))
y=$(( $RANDOM / 100 ))
s=$(($x+$y))
di=$(date "+%s")
echo -n "Qual a soma de $x com $y ? "
read g
while [[ "$g" -ne $s ]]; do
  echo -n "Tente de novo: "
  read g
done
df=$(date "+%s")
t=$(($df-$di))
echo "Acertou em $t segundos"
```

4.8.2 - Listar os ficheiros executáveis da directoria actual

4.8.3 - Compilar todos os ficheiros .c numa dada directoria

4.8.4 - Jogo da forca

Considere o ficheiro words.txt, com o seguinte conteúdo

words.txt

```
barbatana
camelo
carro
caramelo
```

./forca.sh

```
#!/bin/bash

#readc: lê um caracter
readc() {
    echo -n "letra ( dispõe de $ntry tentativas ) : "
    read c
}

#marca : marca o caracter lido
marcac() {
    newd=""
    ok=0
    tryok=1
    for (( i=0; i < n ; i++)) ; do
      sx=${s:i:1}
      dx=${d:i:1}
      if [ $sx = $c ] || [ $dx != "-"  ] ; then
          newd="$newd$sx"
      else
          newd="$newd-"
```

```
                ok=1
        fi
        if [ $sx = $c ] && [ $dx = "-" ] ; then
                tryok=0
        fi
    done
    d=$newd
    ntry=$(( $ntry - $tryok ))
}

#getword : obtém uma palavra do ficheiro
getword() {
    wn=`cat words | wc -l`
    wx=$(( 1 + $wn * $RANDOM / 2**15 ))
    s=`cat words | head -$wx | tail -1`
}

#main
#s=barbatana
getword

n=${#s}
d=""
for (( i=0; i < n ; i++)) ; do
    d="$d-"
done
echo $d

ok=1
ntry=7
while [ ! $ok -eq 0 ] && [ $ntry -gt 0  ] ; do
    readc
    marcac
    echo $d
done
if [ $ok -eq 0 ] ; then
    echo "GANHOU."
else
    echo "PERDEU."
fi
```

5. Ambiente

6. Exemplos

Anexo A - Folha Resumo