



Instituto Superior de Engenharia do Porto  
Departamento de Engenharia Informática

# SISTEMAS OPERATIVOS I

## Parte II

Março de 2006

Berta Batista  
Luis Lino Ferreira  
Maria João Viamonte  
Nuno Malheiro

Sugestões e participações de erros para:

[bbatista@dei.isep.ipp.pt](mailto:bbatista@dei.isep.ipp.pt)

[viamonte@dei.isep.ipp.pt](mailto:viamonte@dei.isep.ipp.pt)

## **AGRADECIMENTOS**

Gostaríamos de agradecer ao colega Orlando Sousa que permitiu a utilização do material pedagógico que serviu de base à elaboração deste texto.

## **BIBLIOGRAFIA**

“UNIX For Application Developers” de William A. Parrete, ISBN 007031697X, Editora McGraw-Hill, June 1, 1991

Advanced Bash-Scripting Guide – Mendel Cooper  
<http://www.tldp.net/LDP/abs/html/index.html>

## INDICE – Programação em BASH

AGRADECIMENTOS .....	2
BIBLIOGRAFIA .....	2
1 PROGRAMAÇÃO EM BASH .....	4
1.1 CRIAÇÃO DE UM SCRIPT .....	4
1.2 VARIÁVEIS .....	5
1.2.1 O USO DE METACARACTERES DE “DISFARCE” .....	6
1.2.2 VARIÁVEIS DE AMBIENTE .....	7
1.2.3 PARÂMETROS .....	7
1.3 EXECUÇÃO DE COMANDOS .....	8
1.4 CONDIÇÕES .....	8
1.4.1 O COMANDO test E O COMANDO [ ] .....	8
1.5 ESTRUTURAS DE CONTROLO .....	10
1.5.1 A ESTRUTURA “if” E “elif” .....	10
1.5.2 O ciclo “for” .....	11
1.5.3 O CICLO “while” .....	11
1.5.4 O CICLO “until” .....	12
1.5.5 A ESTRUTURA “case” .....	12
1.6 LISTAS DE COMANDOS .....	13
1.7 FUNÇÕES; VARIÁVEIS LOCAIS .....	14
1.8 MAIS COMANDOS UNIX (BASTANTE ÚTEIS EM SHELL SCRIPTS) .....	15
1.8.1 break .....	15
1.8.2 continue .....	16
1.8.3 exit .....	16
1.8.4 clear .....	16
1.8.5 export .....	16
1.8.6 expr .....	17
1.8.7 printf .....	17
1.8.8 set .....	18
1.8.9 shift .....	18
1.9 EXPANSÃO DE PARÂMETROS .....	18

# 1 PROGRAMAÇÃO EM BASH

Reunindo técnicas de redireccionamento de input e output e técnicas de programação podem ser criados pequenos programas a que habitualmente se chamam *shell scripts*.

O conhecimento destas técnicas é essencial para quem queira ser, por exemplo, um administrador de um sistema operativo UNIX, já que permite evitar a execução de tarefas repetitivas ou automatizar procedimentos rotineiros e manuais.

A programação de *shell scripts* não é difícil já que se baseia num conjunto não muito extenso de operadores e opções, sendo no resto em tudo semelhante à execução de comandos em UNIX que já foram apresentados.

Resulta pois que a programação da *shell* tem regras (semânticas) diferentes consoante a *shell* que se estiver a utilizar. Neste caso as técnicas e os exemplos que a seguir se apresentam dizem respeito à BASH.

## 1.1 CRIAÇÃO DE UM SCRIPT

Para criar um *script*, é necessário utilizar um editor de texto onde vamos escrever a sequência de comandos que se pretende executar. Considere que o ficheiro **fich1** contém o seguinte:

```
#Cleanup
#Run as root, of course!

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Logs cleaned up"
```

Os comentários começam com **#** e continuam até ao fim da linha.

Este *script* permitia guardar dentro do directório `/var/log`, em dois ficheiros diferentes (`messages` e `wtmp`), o conteúdo do ficheiro `null` que está no directório `/dev`.

Esta é uma tarefa rotineira do administrador se pretender salvaguardar as ocorrências do sistema (*log files*). A execução desta *script* permite automatizar esta tarefa.

Para executar um *script*, podemos utilizar duas formas diferentes. A primeira é invocar a *shell* com o nome do *script* como parâmetro.

```
$ /bin/bash fich1
```

A outra maneira de executar *scripts* é escrever apenas o nome do *script* que queremos executar, garantindo previamente que o ficheiro tem permissão de *execução*.

```
$ chmod +x fich1
$ fich1
```

Para tornar este *script* disponível para todos os utilizadores teríamos que o mover para junto dos restantes utilitários (por ex: nos directórios `/bin` ou `/usr/local/bin`) e seria invocado digitando simplesmente o seu nome.

## 1.2 VARIÁVEIS

Vejam agora uma versão melhorada do *script* apresentado anteriormente

```
#!/bin/bash
# Proper header for a bash script
# Cleanup, version 2
#Run as root, of course!
#Insert code here to print error messages and exit if not root.

LOG_DIR=/var/log
# Variables are better than hard-coded values.

cd $LOG_DIR

cat /dev/null > messages
cat /dev/null > wtmp

echo "Logs cleaned up"

exit #The right and proper method of "exiting" from a script.
```

O comentário **#!/bin/bash** é um comentário especial, onde **#!** informa o sistema que o argumento que se segue é o programa que vai ser utilizado para executar este ficheiro (neste caso está localizado em /bin/bash). Poderá ser omitido se só se usarem comandos genéricos (como no primeiro exemplo apresentado) sem directivas específicas da *shell*.

Como se pode ver no exemplo anterior, para utilizar variáveis não é necessário declará-las primeiro. As variáveis são criadas quando se utilizam, isto é, quando lhes atribuímos um valor inicial.

Por defeito, todas as variáveis são consideradas do tipo *string*, mesmo aquelas que têm valores numéricos. É necessário ter em conta que ao serem do tipo *string* as variáveis são *case-sensitive*.

Deve distinguir-se claramente entre o nome de uma variável e o seu valor. Se **x1** é o nome da variável, então **\$x1** é uma referência ao seu valor.

Pode-se recorrer ao comando **echo** para mostrar esse valor no monitor.

Para se atribuir um valor a uma variável pode-se:

- Atribuir a uma variável o valor de outra - *variable substitution*
- Atribuir um valor recorrendo ao operador de atribuição (= sem espaço nem antes nem depois do operador) – *variable assignment*
- Atribuir um valor a uma variável através do comando **read**

Exemplos:

```
-> a=375
-> hello=$a
-> echo $hello
375

-> valor=Sistemas
-> echo $valor
Sistemas

-> valor="Sistemas Operativos I"
-> echo $valor
Sistemas Operativos I
```

```
-> valor=4+5
-> echo $valor
4+5

-> read num
5
-> echo $num
5
```

```
var1="Hello world"
field=1
delim=':'
filenm=/etc/passwd
echo $var1
Hello world
cut -f$field -d"$delim" $filenm
root
bin
adm
i900000
joe $filenm
...
"/etc/passwd" [read only] 4 lines, 67 characters
```

### 1.2.1 O USO DE METACARACTERES DE "DISFARCE"

Como se sabe existem metacaracteres com significado especial para a *shell*, nomeadamente o espaço e o \$.

Relembre os metacaracteres de disfarce, pág. 28 da Parte I, e que apenas os caracteres \$, ` e \ mantêm o seu significado entre aspas.

O *script*

```
#!/bin/bash

valor="Sistemas Operativos I"

echo $valor
echo "$valor"
echo '$valor'
echo \ $valor

echo Introduza texto:
read valor

echo '$valor' foi modificado para $valor
```

dá como resultado:

```
Sistemas Operativos I
Sistemas Operativos I
$valor
$valor
Introduza texto:
Exame de SO1
$valor foi modificado para Exame de SO1
```

## 1.2.2 VARIÁVEIS DE AMBIENTE

Como já se falou existem variáveis de ambiente que podem ser utilizadas na programação com *shell script*. Os nomes das variáveis ambientes são escritos em maiúsculas para se distinguirem das variáveis definidas nos *scripts* (que normalmente são escritas em minúsculas). A tabela seguinte apresenta algumas variáveis de ambiente:

Variável Ambiente	Descrição
\$HOME	Directório <i>home</i> do utilizador
\$PATH	Lista de directórios separados por :
\$PS1	Prompt (normalmente é \$)
\$PS2	Prompt secundário (normalmente é >)
\$IFS	Input Field Separator. Lista de caracteres utilizada para separar palavras.
\$0	Nome da <i>shell script</i>
\$\$	PID ( <i>Process IDentification</i> ) da <i>shell script</i>
\$PWD	Directório corrente
\$OLDPWD	Pai do Directório corrente

## 1.2.3 PARÂMETROS

Se o *script* é invocado com parâmetros existem mais algumas variáveis que podem ser referenciadas no programa. As variáveis que permitem manipular os parâmetros são:

Variável	Descrição
\$#	Número de parâmetros da <i>shell script</i>
\$1, \$2, \$3, ...	Os parâmetros da <i>script</i>
\$*	Lista com todos os parâmetros, numa única variável, separados pelo primeiro carácter da variável ambiente <b>IFS</b>
\$@	Semelhante ao \$*, só que não utiliza a variável ambiente <b>IFS</b>

Mesmo que não se passem parâmetros, a variável \$# continua a existir, mas obviamente com o valor 0.

Considere que o nome do *script* seguinte é **script\_variaveis** :

```
#!/bin/bash

valor="Sistemas"
echo $valor
echo "O programa $0 está a ser executado"
echo "O segundo parâmetro é $2"
echo "O primeiro parâmetro é $1"
echo "O terceiro parâmetro é $3"
echo "A lista de parâmetros é $*"

```

Se executar o *script*, obtém o seguinte:

```
$ script_variaveis exame sistemas operativos
Sistemas
O programa script_variaveis está a ser executado
O segundo parâmetro é sistemas
O primeiro parâmetro é exame
O terceiro parâmetro é operativos
A lista de parâmetros é exame sistemas operativos

```

## 1.3 EXECUÇÃO DE COMANDOS

É naturalmente interessante poder executar dentro de um *script* os comandos Unix que já conhecemos. Para executar comandos utiliza-se uma técnica (que é naturalmente passível de realizar pela linha de comandos) invocando o comando a ser executado da seguinte forma:

```
$(comando) ou `comando`
```

O resultado é a saída do respectivo comando.

Considere o *script* que mostra o conteúdo da variável `PATH`, bem como os utilizadores que estão actualmente no sistema.

```
#!/bin/bash
echo "A variável PATH é $PATH"
echo "Os utilizadores que estão no sistema são $(who)"
```

O conceito de colocar o resultado da execução de um comando numa variável é muito poderoso, tal como se mostra nos exemplos seguintes:

```
users = `who|wc -l`
echo ` *** There are $users users logged in the system ***`
*** There are $users users logged in the system ***

echo ` *** There are `who | wc -l` users logged in the system ***`
*** There are 5 users logged in the system ***

echo ` *** There are $users users logged in the system ***`
*** There are 5 users logged in the system ***
```

Este mesmo conceito permite "incrementar" uma variável (para implementar um contador, por exemplo).

```
valor=1
valor=$(( $valor + 1 ))
echo $valor
2
```

## 1.4 CONDIÇÕES

Um dos factores essenciais em todas as linguagens de programação é a possibilidade de testar condições e fazer determinadas **acções** em função dessas **condições**.

### 1.4.1 O COMANDO `test` E O COMANDO `[ ]`

Estes dois comandos servem para testar condições e são equivalentes.

Para perceber a funcionalidade do comando `test`, vamos verificar se o ficheiro `file.c` existe, e em caso afirmativo apresentar o seu conteúdo.

```
#!/bin/bash
if test -f fich.c
then
    more fich.c
fi
```

Também podemos utilizar o comando [ ] para obter a mesma funcionalidade. Note que na utilização do comando [ ] é necessário existir um espaço depois de [ , e um espaço antes de ] .

```
#!/bin/bash
if [ -f fich.c ]
then
    more fich.c
fi
```

Apresentam-se de seguida os testes mais comuns:

### Comparação de *strings*

Comparação	Resultado
String	<b>Verdade</b> , se a <i>string</i> não é vazia
String1 = string2	<b>Verdade</b> , se as <i>strings</i> são iguais
String1 != string2	<b>Verdade</b> , se as <i>strings</i> são diferentes
-n string	<b>Verdade</b> , se a <i>string</i> não é <b>nula</b>
-z string	<b>Verdade</b> , se a <i>string</i> é nula

### Comparações aritméticas

Comparação	Resultado
Expressão1 -eq expressão2	<b>Verdade</b> , se forem iguais
Expressão1 -ne expressão2	<b>Verdade</b> , se as expressões são diferentes
Expressão1 -gt expressão2	<b>Verdade</b> , se expressão1 > expressão2
Expressão1 -ge expressão2	<b>Verdade</b> , se expressão1 ≥ expressão2
Expressão1 -lt expressão2	<b>Verdade</b> , se expressão1 < expressão2
Expressão1 -le expressão2	<b>Verdade</b> , se expressão1 ≤ expressão2
!expressão	Nega a expressão. <i>Retorna Verdade</i> se a expressão é <b>falsa</b>

### Condições em ficheiros

Comparação	Resultado
-d ficheiro	<b>Verdade</b> , se o directório existe
-f ficheiro	<b>Verdade</b> , se o ficheiro existe
-r ficheiro	<b>Verdade</b> , se é possível ler o ficheiro
-s ficheiro	<b>Verdade</b> , se o ficheiro tem um tamanho > 0
-w ficheiro	<b>Verdade</b> , se é possível <i>escrever</i> no ficheiro
-x ficheiro	<b>Verdade</b> , se é possível <i>executar</i> o ficheiro

### Operadores lógicos

Operador	Resultado
condição1 && condição2	<b>Verdade</b> , se condição1 e condição2 verdadeiras
condição1    condição2	<b>Verdade</b> , se uma das condições for verdadeira

## 1.5 ESTRUTURAS DE CONTROLO

### 1.5.1 A ESTRUTURA "if" E "elif"

`if` testa o resultado de um comando e executa condicionalmente um grupo de comandos.

```
if condição
then
    comando1
    comando2
    ...
    comandon
else
    comando1
    ...
    comandon
fi
```

**Nota:** Se quiser utilizar o `then` na mesma linha do `if` é necessário acrescentar `;` depois da condição.

Considere o seguinte exemplo que faz uma decisão baseado numa resposta:

```
#!/bin/bash

echo "Passou no exame? "
read resposta

if [ $resposta = "sim" ]; then
    echo "Parabéns!"
else
    echo "Não estudou !!!"
fi
```

O *script* anterior tem um problema, escreve "Não estudou" para qualquer resposta, excepto a resposta sim. Para resolver esta situação podemos utilizar o comando `elif`, que permite testar uma segunda condição quando o `else` é executado.

```
#!/bin/bash

echo "Passou no exame? "
read resposta

if [ $resposta = "sim" ]; then
    echo "Parabéns!"
elif [ $resposta = "não" ]; then
    echo "Não estudou !!!"
else
    echo "Não conheço a resposta $resposta. Introduza sim ou não!"
fi
```

## 1.5.2 O ciclo "for"

Esta instrução executa um ciclo um determinado número de vezes (em função de um conjunto de valores).

```
for variável in valores
do
    comando 1
    ...
    comando n
done
```

Considere o seguinte exemplo:

```
#!/bin/bash

for valor in exame de sistemas "SO1 - teste" operativos
do
    echo $valor
done
```

O resultado é:

```
exame
de
sistemas
SO1 - teste
operativos
```

Analise agora o seguinte exemplo:

```
#!/bin/bash

for valor in $(ls so[123].txt)
do
    more $valor
done
```

Este exemplo mostra o conteúdo dos ficheiros que são o resultado de executar `ls so[123].txt`, isto é, mostra o conteúdo dos ficheiros `so1.txt`, **`so2.txt`** **`so3.txt`** se existirem.

Como se sabe o ciclo `for` funciona bem quando se trata de situações em que sabemos o número exacto de vezes que pretendemos que ele se repita. Quando é necessário executar um grupo de comandos um número **variável de vezes**, este ciclo não é o mais aconselhado.

## 1.5.3 O CICLO "while"

O ciclo `while` é útil nas situações em que não existe um número fixo de vezes para executar um determinado grupo de comandos.

```
while condição
do
    comando 1
    ...
    comando n
done
```

Considere o *script*:

```
#!/bin/bash

echo "Introduza um nome: "
read nome

while [ "$nome" != "Sistemas" ]; do
    echo "Não acertou no nome - tente de novo !"
    read nome
done
```

Este *script* só termina quando o utilizador introduzir o nome correcto (Sistemas). Enquanto introduzir o nome errado o ciclo vai ser repetido.

**Nota:** A utilização de aspas (") em [ "\$nome" != "Sistemas" ] permite salvaguardar a situação em que o utilizador utiliza o **Enter** sem introduzir mais nada.

Nesta situação a condição de teste ficaria [ != "Sistemas" ], que não é uma condição válida.

Com a utilização de aspas o problema é resolvido, pois a condição de teste será [ "" != "Sistemas" ].

#### 1.5.4 O CICLO "until"

O ciclo `until` é semelhante ao ciclo `while`. A única diferença é que o ciclo `while` continua enquanto uma condição é verdadeira e o ciclo `until` continua até que a condição seja verdade.

```
until condição
do
    comando 1
    ...
    comando n
done
```

Considere o *script*:

```
#!/bin/bash

until who | grep "$1" >/dev/null
do
    sleep 10
done

echo *** O utilizador $1 entrou no sistema ! ***
```

Este *script* verifica se um determinado utilizador entrou no sistema, isto é, de 10 em 10 segundos verifica se o utilizador está no sistema. Quando o utilizador entrar no sistema o *script* termina.

#### 1.5.5 A ESTRUTURA "case"

Esta estrutura permite verificar o conteúdo de uma variável em relação a vários **padrões**, executando depois os respectivos comandos.

```
case variável in
    padrão [| padrão ...]) comandos;;
    padrão [| padrão ...]) comandos;;
    ...
esac
```

Considere o *script*:

```
#!/bin/bash

echo "Passou no exame? "
read resposta

case "$resposta" in
    "sim") echo "Parabéns!" ;;
    "não") echo "Não estudou !!!" ;;
    "s" ) echo "Parabéns!" ;;
    "n" ) echo "Não estudou !!!" ;;
    *    ) echo "Não conheço a resposta $resposta!" ;;
esac
```

O *script* compara o conteúdo de `resposta` com todos os `padrões` (quando se verifica um dos `padrões` o comando `case` termina a procura).

O asterisco (\*) pode ser utilizado para expandir *strings*. Neste exemplo, o asterisco faz concordância (*matching*) de todas as *strings*, permitindo assim executar uma acção por defeito (quando nenhum dos outros padrões se verificou).

Obtemos a mesma funcionalidade com o *script*:

```
#!/bin/bash

echo "Passou no exame? "
read resposta

case "$resposta" in
    "sim" | "s" ) echo "Parabéns!" ;;
    "não" | "n" ) echo "Não estudou !!!" ;;
    *      ) echo "Não conheço a resposta $resposta!" ;;
esac
```

## 1.6 LISTAS DE COMANDOS

Para executar uma lista de comandos em que só é necessário executar o comando seguinte se o comando anterior foi bem sucedido, isto é, o `comando2` só é executado se o `comando1` teve sucesso, o `comando3` só é executado se o `comando2` teve sucesso, etc, faz-se o seguinte:

```
comando1 && comando2 && comando3 && ...
```

Para executarmos uma série de comandos até que um tenha sucesso, isto é, se o `comando1` tem sucesso, já não é executado mais nenhum comando da lista; se o `comando1` falhou, então é executado o `comando2`; se o `comando2` tem sucesso então termina; se o `comando2` falhou então é executado o `comando3`, e assim sucessivamente, faz-se o seguinte:

```
comando1 || comando2 || comando3 || ...
```

exemplo:

```
#!/bin/bash

rm $TEMPDIR/* && echo "Files successfully removed"

rm $TEMPDIR/* || echo "Files were not removed"
```

## 1.7 FUNÇÕES; VARIÁVEIS LOCAIS

As funções têm a seguinte estrutura:

```
nome_da_função () {
    comandol
    ...
    comandon
}
```

Considere o *script*:

```
#!/bin/bash

escreve () {
    echo "A função está a ser executada "
}

echo "Início do script"
escreve
echo "Fim do script"
```

É necessário definir a função antes de a utilizar, isto é, o código das funções deve ser colocado no princípio do *script*. Quando uma função é invocada, os parâmetros do *script* `*$`, `$@`, `$#`, `$1`, `$2`, etc, são substituídos pelos parâmetros da função. Quando a função termina a sua execução, os parâmetros são restaurados.

Para declarar variáveis locais à função, utiliza-se a palavra **local** antes da variável. Uma variável também pode ser declarada como apenas de leitura utilizando a palavra **readonly** antes da variável.

**Nota:** Contrariamente ao que acontece na linguagem C, na programação em *bash* uma variável declarada dentro de uma função só é local se for declarada como tal.

Considere o *script*:

```
#!/bin/bash

texto="Variável global"

escreve () {
    local texto="Variável local"
    echo "A função está a ser executada"
    echo $texto
}

echo "Início da script"
echo $texto
escreve
echo $texto
echo "Fim da script"
```

Este *script* dá o seguinte resultado:

```
Início da script
Variável global
A função está a ser executada
Variável local
Variável global
Fim da script
```

Para que a função retorne um **valor numérico**, é necessário utilizar o comando **return**. Quando não se utiliza o comando **return** na função, a função retorna o estado do último comando que foi executado.

A única maneira de retornar *strings* é utilizar uma **variável global**, de modo a ser possível utilizá-la quando a função terminar a sua execução.

Considere o *script* teste que **passa** parâmetros para a função e em que a função retorna valores numéricos:

```
#!/bin/bash

pergunta() {
    echo "Os parâmetros da função são $*"
    while true
    do
        echo -n "sim ou não"
        read resposta
        case "$resposta" in
            s | sim ) return 0;;
            n | não ) return 1;;
            * ) echo "Responda sim ou não"
        esac
    done
}

echo "Os parâmetros da script são $*"

if pergunta "O nome é $1 ?"
then
    echo "Olá $1"
else
    echo "Engano"
fi
```

Exemplo de utilização da *script* anterior:

```
$ teste Orlando Sousa
Os parâmetros da script são Orlando Sousa
Os parâmetros da função são O nome é Orlando?
sim ou não
não
Engano
```

## 1.8 MAIS COMANDOS UNIX (BASTANTE ÚTEIS EM SHELL SCRIPTS)

### 1.8.1 break

Este comando é utilizado para sair de um ciclo `for`, `while` ou `until`.

```
#!/bin/bash
# Esta script mostra o nome do primeiro directório com iniciais so

for ficheiro in so*
do
    if [ -d "$ficheiro" ]; then
        break;
    fi
done
```

```
echo O primeiro directório com iniciais so é $ficheiro
```

### 1.8.2 continue

Este comando permite avançar para a próxima iteração do ciclo `for`, `while` ou `until`.

```
#!/bin/bash
# Esta script apenas mostra os nomes de ficheiros que tenham como
# iniciais so (não mostra os directórios)

for ficheiro in so*
do
    if [ -d "$ficheiro" ]; then
        continue
    fi
    echo $ficheiro
done
```

### 1.8.3 exit

Este comando tem duas finalidades:

- parar a execução do *shell script*
- retornar um estado para a execução desse mesmo shell script, se especificarmos um valor não nulo (verificável através da variável `$?`)

### 1.8.4 clear

Trata-se de um comando que permite limpar o monitor tipo *clearscreen*.

### 1.8.5 export

O comando `export` faz que uma variável fique **visível**, isto é, cria uma variável ambiente que fica disponível para eventuais *sub-shells* que o utilizador crie e que desaparece quando se faz *logout*. Não é possível contudo fazer com que as variáveis de um utilizador sejam disponibilizadas a outro.

Considere os seguintes *scripts*:

teste2:

```
#!/bin/bash

echo $valor
echo $resposta
```

teste1:

```
#!/bin/bash

valor="Variável que não utiliza export"
export resposta="Variável que utiliza export"

teste2
```

Se executarmos o *script* `teste1`, dá:

```
Variável que utiliza export
```

Apenas é visível a variável **resposta** no *script teste2* (que foi exportada) e não a variável **valor**.

### 1.8.6 expr

O comando `expr` avalia argumentos de uma expressão. É normalmente utilizado para cálculos aritméticos. A sua forma genérica é:

```
expr integer operator integer
```

Algumas questões importantes:

- o operador `*` para indicar multiplicação tem de ser "disfarçado" (`expr 4 * 5` dá erro).
- as operações de multiplicação e divisão têm precedência sobre a soma e a subtração.
- os parêntesis não são reconhecidos pelo comando `expr` pelo que mudar a precedência das operações tem de ser feito "manualmente".
- o resultado de uma divisão é sempre um inteiro.

Alguns exemplos:

```
expr 4 \* 5
20
expr 5 + 7 / 3
7
int = `expr 5 + 7 `
expr $int / 3
4
expr `expr 5 + 7` / 3
4
```

Expressão	Descrição
Expressão1   expressão2	Expressão1, se é diferente de zero; senão expressão2
Expressão1 & expressão2	Zero, se uma ou ambas as expressões são zero
Expressão1 = expressão2	Igualdade
Expressão1 != expressão2	Diferentes
Expressão1 > expressão2	
Expressão1 ≥ expressão2	
Expressão1 < expressão2	
Expressão1 ≤ expressão1	
Expressão1 + expressão2	Adição
Expressão1 - expressão2	Subtração
Expressão1 * expressão2	Multiplicação
Expressão1 / expressão2	Divisão inteira
Expressão1 % expressão2	Resto da divisão

### 1.8.7 printf

O comando `printf` é utilizado para formatar a saída. A sintaxe para este comando é:

```
printf "formato da string" parâmetro1 parâmetro2 ...
```

O formato da *string* é semelhante ao formato utilizado na linguagem C, com algumas restrições (só suporta valores inteiros, pois a *shell* faz todas as suas operações sobre valores inteiros).

### 1.8.8 set

O comando `set` permite configurar as variáveis da *shell*.

Considere o *script*:

```
#!/bin/bash
echo A data é $(date)
set $(date)
echo O mês da data é $2
```

O *script* anterior apresenta relativamente ao primeiro `echo` o resultado de executar o comando `date` (ex: Mon Jan 17:22:57 MET 1999) mas apenas o segundo campo (que contém o mês) é apresentado no segundo `echo`.

### 1.8.9 shift

O comando `shift` retira um parâmetro aos parâmetros do *script* e “roda” todos os outros (ex: \$2 torna-se o \$1, o \$3 torna-se o \$2, etc). O `shift` é utilizado para pesquisar os parâmetros passados ao *script*.

O *script* seguinte mostra todos os parâmetros introduzidos na invocação do *shell script*.

```
#!/bin/bash
while [ "$1" != "" ]; do
    echo $1
    shift
done
```

## 1.9 EXPANSÃO DE PARÂMETROS

A expansão de parâmetros é muito útil no manuseamento de partes desses parâmetros.

Suponha que precisa de um *script* que processe o ficheiro **1.tmp** e o **2.tmp**.

O *script* que eventualmente teria escrito seria:

```
#!/bin/bash
for valor in 1 2
do
    processa $i.tmp
done
```

Este *script* não funciona, pois o que a *shell* está a tentar fazer é substituir o valor da variável `$i.tmp`, que não existe.

Para proteger a expansão da variável é necessário que o `i` fique entre `{ }` (relembrar a opção `-exec` do comando `find`...).

O *script* correcto é o seguinte em que em cada iteração o valor de *i* é substituído por `${i}`.

```
#!/bin/bash

for valor in 1 2
do
    processa ${i}.tmp
done
```

Mas a expansão de parâmetros apresenta outras funcionalidades que a mera substituição. Na tabela seguinte sumariaram-se essas funcionalidades.

Expansão de Parâmetros	Descrição
<code>\${parâmetro:-valor}</code>	Se <b>parâmetro</b> é nulo então dá como resultado <b>valor</b> (inicialização)
<code>\${#parâmetro}</code>	Tamanho do parâmetro
<code>\${parâmetro%palavra}</code>	Do fim, remove a parte mais pequena que contenha <b>palavra</b> e <i>retorna</i> o resto.
<code>\${parâmetro%%palavra}</code>	Do fim, remove a parte mais longa que contenha <b>palavra</b> e <i>retorna</i> o resto.
<code>\${parâmetro#palavra}</code>	Do inicio, remove a parte mais pequena que contenha <b>palavra</b> e <i>retorna</i> o resto.
<code>\${parâmetro##palavra}</code>	Do inicio, remove a parte mais longa que contenha <b>palavra</b> e <i>retorna</i> o resto.

Exemplo:

```
#!/bin/bash

echo ${valor:-Vazio}
valor=Cheio
echo ${valor:-Vazio}
valor=/usr/bin/X11/startx
echo ${valor#*/}
echo ${valor##*/}

valor=/usr/local/etc/local/networks
echo ${valor%local*}
echo ${valor%%local*}
```

dá como resultado:

```
Vazio
Cheio
usr/bin/X11/startx
startx
/usr/local/etc/
/usr/
```

O *script* seguinte muda todos os ficheiros cujo nome termina em **.txt** para **.doc**:

```
#!/bin/bash
for ficheiro in *.txt
do
    mv $ficheiro ${ficheiro%.txt}.doc
done
```