Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

Programação em *Shell Script*

Princípios da Computação
2006/2007

Novembro de 2006

Luis Lino Ferreira
Luis Nogueira
Berta Batista
Bertil Marques
Maria João Viamonte
Nuno Malheiro

Sugestões e participações de erros para:
llf@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

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 Linux, 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 Linux 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 com o nome **fich1** contém o seguinte texto:

```
#O meu primeiro script

echo "Olá mundo!"
date
```

Os comentários começam com **#** e continuam até ao fim da linha.
Este *script* utiliza o comando **echo** para imprimir a frase "Olá mundo", de seguida imprime a data e termina.
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 u+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.
Se o directório actual (o directório '.') não estiver pertencer ao seu PATH, terá que executar os ficheiros utilizando com comando ./nome_do_script. No DEI o directório actual normalmente não pertence ao PATH. Para ver o seu PATH (os directórios onde sistema operativo procura os comandos digitados na linha de comandos) pode executar o comando seguinte: echo $PATH.

1.2 VARIÁVEIS

Vejamos agora uma outra versão do *script* apresentado anteriormente

```
#!/bin/bash

a="Olá Mundo"
echo $a
date
exit
```

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 regra, 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 **a** é o nome da variável, então **$a** 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*
- Atribuír 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 (podem correr na linha de comando):

```
$ a=375
$ hello=$a
$ echo $hello
375

$ valor=Principios
$ echo $valor
Principios

$ valor=”Princípios da Computação”
$ echo $valor
Princípios da Computação

$ valor=4+5
$ echo $valor
4+5

$ read num
5
$ echo $num
5
```

Outros exemplo:

```
$ 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
```

Atenção, que o resultado de correr este último exemplo no Slax ou nos servidores do departamento deverá ser diferente.

1.2.1 O USO DE METACARACTERES DE "DISFARCE"

Como se sabe existem metacaracteres com significado especial para a *shell*, nomeadaemte o espaço e o $. Relembre os metacarectes de disfarce, pág. 28 da Parte I, e que apenas os caracteres **$**, **`** e **** mantêm o seu significado entre aspas.
O *script*

```
#!/bin/bash

valor="Princípios da Computação"

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

echo Introduza texto:
read valor

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

dá como resultado:

```
Princípios da Computação
Princípios da Computação
$valor
$valor
Introduza texto:
Exame de Princípios da Computação
$valor foi modificado para Exame de Princípios da Computação
```

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 *home* 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.
$$	PID (*Process IDentification*) da *shell script*
$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 *shell script*
$0	Nome da *shell script*
$1, $2, $3, ...	Os parâmetros da *script*
$*	Lista com todos os parâmetros, numa única variável, separados pelo primeiro caracter da variável ambiente **IFS**
$@	Semelhante ao **$***, só que não utiliza a variável ambiente **IFS**

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 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 ' Existem $users utilizadores activos no sistema '
Existem $users utilizadores activos no sistema

$echo " Existem `who | wc –l` utilizadores activos no sistema"
Existem 5 utilizadores activos no sistema

$echo "Existem $users users utilizadores activos no sistema
Existem 5 utilizadores activos no sistema
```

Este mesmo conceito permite "incrementar" uma variável (para implementer 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 **`fich.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 `]`**, caso contrário a shell não consegue interpretar este comando.

```
#!/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	**Verdade**, se a *string* não é vazia
String1 = string2	**Verdade**, se as *strings* são iguais
String1 != string2	**Verdade**, se as *strings* são diferentes
-n string	**Verdade**, se a *string* não é **nula**
-z string	**Verdade**, se a *string* é nula

Comparações aritméticas

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

Condições em ficheiros

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

Operadores lógicos

Operador	Resultado
condição1 && condição2	**Verdade**, se condição1 e condição2 verdadeiras
condição1 || condição2	**Verdade**, 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 "Parabens!"
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 omissão (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 e VARIÁVEIS LOCAIS

As funções têm a seguinte estrutura:

```
nome_da_função () {

      comando1
      …
      …
      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 $*"
      while true
            echo -n "sim ou não"
            read resposta
            case "$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 (BASTANTE ÚTEIS EM SHELL SCRIPTS)

1.8.1 break

Este comando é utilizado para saír de um ciclo **for**, **while** ou **until**.

```
#!/bin/bash
# Esta script mostra o nome do primeiro directório cujas letras iniciais são
"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 subtracção.
- os parentesis não são reconhecidos pelo comando **expr** pelo que mudar a precedência das operações tem de ser feito pelo utilizador.
- o resultado de uma divisão é sempre um inteiro.
- os espaços entre operadores devem ser sempre respeitados

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	Subtracção
Expressão1 * expressão2	Multiplicação
Expressão1 / expressão2	Divisão inteira
Expressão1 % expressão2	Resto da divisão

Pode-se também utilizar a expressões do tipo $((expressão aritmética)) para efectuar cálculos. Alguns exemlos:

```
$ echo $((4 + 5))
9
$ a=20
$ echo $((a+2))
22
```

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
```