2 O SHELL COMO UMA LINGUAGEM DE PROGRAMAÇÃO

Existem duas maneiras de escrever programas diretamente no *shell*. Ou o usuário digita diretamente os comandos no *prompt*, ou um arquivo contendo os comandos é criado, sendo dada permissão de execução a este arquivo, para posteriormente ser utilizado como um programa comum.

Na primeira forma, o usuário digita os comandos como normalmente o faz. Entretanto, alguns comandos aceitos pelo *shell* normalmente possuem mais de uma linha para estarem completos. Sempre que o *shell* entende que um comando não foi terminado, um *prompt* diferente é apresentado, informando o usuário a necessidade de continuar o comando. Em nosso caso, este *prompt* adicional é o sinal de maior (>), enquanto o *prompt* usual é o sinal de dólar ($), ou cerquilha (#) para o usuário **root**.

Vejamos dois exemplos:

```
$ for arquivo in *
> do
> if grep -l TEXTO $arquivo
> then
> more $arquivo
> fi
> done
arquivo1.txt
Esta eh uma linha contendo a palavra TEXTO. Bom proveito!
$
```

Neste caso, o comando **for … do** não termina enquanto a palavra **done** não for encontrada, indicando o fim do comando **for … do**. Observe também que o comando **if … then** é terminado com a palavra **fi**. O exemplo teria ficado mais claro se identação fosse utilizada:

```
$ for arquivo in *
> do
>    if grep -l TEXTO $arquivo
>    then
>       more $arquivo
>    fi
> done
arquivo1.txt
Esta eh uma linha contendo a palavra TEXTO. Bom proveito!
$
```

Vale notar que o comando **for** é acompanhado de um **do**. Entretanto são comandos separados, e por isso vêm em linhas diferentes. Para tornar mais legível ainda nosso programa, podemos colocar os dois comandos na mesma linha, separados por ponto-e-vírgula, da mesma forma como faríamos no *shell* para executarmos dois comandos em seqüência. O mesmo fazemos com o **if … then**:

```
$ for arquivo in * ; do
>    if grep -l TEXTO $arquivo ; then
>       more $arquivo
>    fi
> done
arquivo1.txt
Esta eh uma linha contendo a palavra TEXTO. Bom proveito!
$
```

Explicando o exemplo, o comando **for** irá atribuir à variável **arquivo** em cada iteração do laço, o nome de cada arquivo existente no diretório corrente, o que é especificado com o asterisco

(*). A cada iteração, o comando **grep –l TEXTO $arquivo** procura pela palavra **TEXTO** no arquivo indicado pelo valor da variável **arquivo**. O valor de uma variável é sempre referenciado colocando-se um sinal de dólar ($) antes do nome da variável. Neste caso, o comando **grep** com a opção **–l** apenas mostra o nome do arquivo que contém o **TEXTO** procurado, e não seu conteúdo. Como o comando **grep** estava dentro de um **if**, caso tenha tido sucesso, isto é, **TEXTO** foi encontrado dentro de **$arquivo**, o comando **more $arquivo** é executado, o que fará com que o conteúdo do arquivo seja mostrado tela a tela.

Em alguns casos, é desejável obter o resultado de um comando, isto é, aquilo que ele mostra na tela, e aplicá-lo a outro comando. Isto pode ser feito de duas maneiras:

```
$ more `grep –l TEXTO *`
Esta eh uma linha contendo a palavra TEXTO. Bom proveito!

  ou ...

$ more $(grep –l TEXTO *)
Esta eh uma linha contendo a palavra TEXTO. Bom proveito!
```

Neste exemplo, o comando **more** irá mostrar na tela o conteúdo do arquivo cujo nome será o resultado do comando **grep –l TEXTO ***. Observe que existem duas formas: ou o comando a ser executado deve ser delimitado por crase, ou ser envolvido por **$(…)**.

O exemplo a seguir mostra a diferença em usar crase (`) ou $( ) envolvendo um comando:

```
$ more $(grep –l TEXTO *)
Esta eh uma linha contendo a palavra TEXTO. Bom proveito!
$ grep –l POSIX * | more
arquivo1.txt
```

No segundo comando, o resultado do comando **grep** (arquivo1.txt) é passado pelo *pipe* para o comando **more**, que irá mostrá-lo tela a tela. No primeiro comando, o comando **more** irá utilizar como parâmetro o resultado do comando **grep**.

2.1 Criando um Script

Para criar um *script* é fácil. Basta criar um arquivo contendo todos os comandos desejados para o *script*, utilizando para isto qualquer editor de textos (p. ex. **vi**):

```
#!/bin/sh

# primeiro.sh
# Este arquivo eh um script que procura por todos os arquivos contendo a
# palavra TEXTO no diretorio atual, e entao imprime estes arquivo em
# stdout

for arquivo in *; do
   if grep -q TEXTO $arquivo; then
      more $arquivo
   fi
done

exit 0
```

Uma vez salvo o arquivo como nome **primeiro.sh**, precisamos torná-lo executável:

```
$ chmod +x primeiro.sh
```

Neste caso demos permissão de execução para todos. Se isto não for feito, o UNIX não terá como executar o arquivo, por causa de suas permissões. Duas observações são necessárias. Primeiro, observe que na primeira linha, **#!/bin/sh**, está sendo informado ao sistema operacional, qual programa irá interpretar os comandos que seguem, já que o conteúdo do arquivo não é um programa binário executável do sistema. Quando o UNIX encontra os caracteres **#!** nos dois primeiros *bytes* de um arquivo, ele trata o restante da linha como a localização do programa que irá interpretar o restante do *script*. Programas executáveis possuem outros *bytes* que dizem ao sistema operacional o tipo do executável. Isto garante ao UNIX uma forma de "reconhecer" vários tipos de executáveis.

A segunda observação é que as linhas que começam com cerquilha (#) são tratadas como comentários. Somente a primeira linha, se possuir o caractere cerquilha seguido de uma exclamação (#!), é que será tratada de forma diferente.

A única diferença para o nosso exemplo anterior é que incluímos o comando **exit 0** no final do *script*. Apesar de não ser obrigatório, é boa prática colocá-lo indicando aos programas que invocarem nosso *script* se a execução foi bem sucedida ou não. No UNIX, quando um programa retorna zero (0), ele FOI bem sucedido. Quando retorna algo diferente de zero, ele NÃO FOI bem sucedido. Este é o método que o comand **if** do nosso *script* utiliza para saber se o comando **grep** foi bem sucedido ou não. O programa **grep** foi construído para retornar zero quando consegue encontrar a palavra especificada, e diferente de zero quando não conseguir.

Pode acontecer de ao tentarmos executar nosso *script*, recebermos uma mensagem de que ele não existe:

```
$ primeiro.sh
bash: primeiro.sh: command not found
```

Isto acontece porque normalmente o diretório corrente não faz parte do caminho de procura por executáveis, definido pela variável de ambiente **PATH**. A solução é incluir o diretório atual, representado pelo ponto (.) no **PATH**, ou então dizermos explicitamente que queremos executar o *script* no diretório corrente:

```
$ ./primeiro.sh
Esta eh uma linha contendo a palavra TEXTO. Bom proveito!
# palavra TEXTO no diretorio atual, e entao imprime estes arquivo em
```

Curiosamente, nosso *script* também encontrou a palavra TEXTO existente nos comentários do nosso próprio *script*, por isso foi mostrada!

3 SINTAXE DO SHELL

Após os exemplos iniciais, é hora de entrarmos em maiores detalhes do poder de programação do *shell*.

O *shell* é particularmente fácil de aprender, não apenas porque a linguagem é fácil, mas porque é possível construir pequenos fragmentos do *script* separadamente, e mais tarde uní-los em um único *script* mais poderoso.

3.1 Variáveis

Não é necessário declarar variáveis no *shell* antes de usá-las. Elas simplesmente são criadas após o primeiro uso, e destruídas após o *shell* terminar. Todas as variáveis são na verdade do tipo *string*, mesmo que valores numéricos tenham sido atribuídos a elas. Na verdade, uma *string* com os números é atribuída a variáveis quando se deseja trabalhar com valores numéricos. O *shell* e outros utilitários irão converter a *string* para um valor numérico quando for necessário usá-los.

Outro ponto importante é que os nomes das variáveis são sensíveis ao contexto, isto é, a variável **teste** é diferente de **Teste**, e também diferente de **TESTE**, assim como das outras combinações possíveis.

Como dissemos anteriormente, o conteúdo de uma variável é acessado colocando-se o símbolo dólar ($) antes do seu nome. Para vermos seu conteúdo, podemos utilizar o comando **echo**:

```
$ saudacao=Oi
$ echo saudacao      → irá imprimir a palavra "saudacao"
saudacao
$ echo $saudacao     → irá imprimir o conteúdo da variável "saudacao"
Oi
$ saudacao="Ola mundo"     → se a string contem espacos, limitar por aspas ou apostrofos
$ echo $saudacao
Ola mundo
$ saudacao=7+5       → valores numericos sao considerados strings
$ echo $saudacao
7+5
```

Outra forma de atribuir valor a uma variável é utilizando o comando **read**. Este comando lê caracteres do teclado até que a tecla ENTER seja pressionada:

```
$ read var
Estou digitando isso
$ echo $var
Estou digitando isso
```

Uma terceira maneira de atribuir valores a variáveis é utilizando a crase ou $(…), como vimos nos exemplos anteriores:

```
$ var=$(ls -l arquivo1.txt)
$ echo $var
-rwxrwxr-x   1 celso    celso       10147 May 31 01:32 arquivo1.txt
```

Uma última observação sobre variáveis: quando um valor estiver sendo atribuído, o sinal de igual (=), deve se usado entre o nome da variável e o valor correspondente sem espaços.

3.1.1 Delimitando Strings

Vimos que quando *strings* contêm espaços, precisamos delimitá-las com aspas (“) ou apóstrofos (‘). Existe uma diferença entre utilizar um ou outro:

```
$ var=”Oi galera”
$ echo $var
Oi galera
$ echo ”$var”
Oi galera
$ echo ’$var’
$var
$ echo dolar=\$, aspa=\”, apostrofo=\’
dolar=$, aspa=”, apostrofo=’
$ echo ’dolar=\$, aspa=\”, apostrofo=\’\’
dolar=\$, aspa=\”, apostrofo=\’
$ var2=”var=$var”
$ echo $var2
var=Oi galera
$ var2=’var=$var’
$ echo $var2
var=$var
```

Deste exemplo concluímos que usando aspa (“) o *shell* expande nomes de variáveis dentro da *string* para seu valor correspondente, enquanto utilizando apóstrofo (‘), o *shell* trata todo o texto literalmente como foi digitado.

Ao criar *scripts*, a melhor recomendação é sempre testar antes como o *shell* reage aos vários caracteres especiais dentro de uma *string* delimitando-os ou não por aspas ou apóstrofos.

3.1.2 Variáveis de Ambiente e de Parâmetros do Script

Quando *shell script* inicia, algumas variáveis herdadas do *shell* pai são herdadas. Normalmente, as variáveis utilizadas pelo *shell* são todas em letras maiúsculas, para caracterizar variáveis de ambiente. As variáveis utilizadas pelos *scripts* em geral são em letras minúsculas, para diferenciá-las das variáveis definidas pelo próprio *shell*. Principais variáveis de ambiente:

Variável de Ambiente	Descrição
$HOME	O diretório pessoal do usuário corrente
$PATH	Uma lista de diretórios separados por dois-pontos (:) onde o shell irá procurar por comandos
$PS1	O prompt de comandos, normalmente o símbolo dólar ($)
$PS2	O prompt secundário para dados adicionais de um comando, normalmente o símbolo maior (>)
$IFS	Um separador de campos interno do shell. Pode ser uma lista de caracteres. Normalmente quando o shell está lendo dados de stdin, os caracteres espaço, tabulação, e enter são separadores

Quando o *shell script* é invocado, algumas variáveis indicam informações úteis sobre seu nome, os parâmetros que lhe foram passados, e o número do processo que o criou:

Variável de Parâmetro	Descrição
$0	O nome do arquivo correspondente ao *script* chamado, incluindo o caminho de diretório
$1, $2, …	Os argumentos passados para o *script*, onde $1 é o primeiro, e $2 é o segundo, e assim por diante
$#	A quantidade de argumentos passados
$*	Uma lista de todos os parâmetros, separados pelo primeiro caractere da variável IFS
$@	Semelhante ao $*, mas a lista é separada por espaços, desconsiderando o valor de IFS
$$	O número do PID do *shell* que está interpretando o *script*

3.2 Condições

Uma das principais funções em qualquer linguagem de programação é a do teste condicional. Em *shell scripts* isto não é diferente. Freqüentemente é necessário verificar por determinada condição para saber que ação tomar.

Já vimos que o comando **if … then … fi** serve para testar a condição de saída de um programa. Entretanto, é necessário um mecanismos de fazer testes lógicos quaisquer, por exemplo, se uma variável é igual a outra, ou se um um arquivo existe.

O comando **test** implementa os diversos testes lógicos necessários. Assim, o comando **if** testa o resultado do comando **test**. Se foi bem sucedido, executa os comandos dentro do corpo do **if**, caso contrário executa os comandos dentro do corpo do ramo **else** (se existir).

Uma das formas mais simples de uso do comando **test** é o teste da existência de um arquivo. Isto é feito na forma **test –f nome-do-arquivo**. Vejamos um exemplo em um fragmento de *script*:

```
if test -f arquivo1.txt
then
   ...
fi
```

Como o comando **test** normalmente é usado com o comando **if**, um sinônimo para o comando é substituir **test** por colchetes delimitando a condição:

```
if [ -f arquivo1.txt ]
then
   ...
fi
```

É de vital importância não esquecer que deve existir espaço após o abre-colchete ([ ) e antes do fecha-colchete ( ]). Para memorizar esta restrição, lembre-se que um comando sempre precisa vir seguido de espaço. Neste caso, o nome do programa é **“ [“**.

Aproveite este momento e verifique no seu sistema se existe um programa chamado **“ [“**.

Lembre-se que o comando **if** pode ser usado com a cláusula **else**:

```
if [ $var = "Oi galera" ]
then
   ...
else
   ...
```

```
fi
```

Principais comparações do comando **test**:

Comparação de Strings	Resultado
string	Verdadeiro se a *string* não é vazia
string1 = string2	Verdadeiro se as *strings* forem iguais
string1 != string2	Verdadeiro se as *strings* forem diferentes
-n string	Verdadeiro se a string é não nula
-z string	Verdadeiro se a string é nula (vazia)

Quando estiver sendo utilizado o valor de uma variável para comparação como *string*, precedê-la de dólar ($) para obter seu valor. Por segurança, sempre delimitar variáveis por aspas (") para que *strings* nulas não gerem um erro.

Comparação Aritmética	Resultado
expressão1 –eq expressão2	Verdadeiro se as expressões são iguais
expressão1 –ne expressão2	Verdadeiro se as expressões não são iguais
expressão1 –gt expressão2	Verdadeiro se expressão1 é maior que expressão2
expressão1 –ge expressão2	Verdadeiro se expressão1 é maior ou igual que expressão2
expressão1 –lt expressão2	Verdadeiro se expressão1 é menor que expressão2
expressão1 –le expressão2	Verdadeiro se expressão1 é menor ou igual que expressão2
! expressão	! nega a expressão, e retorna verdadeiro se a expressão é falsa

Testes com Arquivos	Resultado
-d arquivo	Verdadeiro se arquivo é um diretório
-e arquivo	Verdadeiro se arquivo existe
-f arquivo	Verdadeiro se arquivo é um arquivo comum
-g arquivo	Verdadeiro se arquivo possui o bit SGID ligado
-r arquivo	Verdadeiro se arquivo pode ser lido (*readable*)
-s arquivo	Verdadeiro se arquivo possui tamanho diferente de zero
-u arquivo	Verdadeiro se arquivo possui o bit SUID ligado
-w arquivo	Verdadeiro se arquivo pode ser escrito (*writeable*)
-x arquivo	Verdadeiro se arquivo pode ser executado (*executable*)

3.3 Estruturas de Controle

O comando **if** é a principal estrutura de controle de *shell scripts*. Além deste, existem algumas estruturas que permitem repetição. Uma delas é o comando **for**, que possui o formato:

```
for variavel in valores
do
    comandos
done
```

A única restrição do **for** em relação ao comando correspondente em uma linguagem de programação tradicional, é que não é possível especificar uma faixa de valores para o **for**, como normalmente ele é utilizado. Assim, para fazer a variável variar de 1 a 5 em uma estrutura de

repetição utilizando o comando **for**, seríamos obrigados a fazer:

```
for i in 1 2 3 4 5; do
   comandos
done
```

Por outro lado, o comando **for** trabalha com listas de forma bastante simples:

```
$ for usuario in $(cut -f1 -d: /etc/passwd)
> do
>    echo $usuario
> done
root
bin
adm
...
```

Um tipo de laço mais adequado a repetições em um certo número conhecido, é o **while**. Interessante é que nas linguagens de programação tradicionais, o **for** é o mecanismo mais adequado a esta situação:

```
while condicao do
   comandos
done
```

Para ilustrar seu uso, vejamos como poderíamos fazer uma repetição para 30 elementos:

```
$ i=1
$ total=30
$ while [ $i -le $total ]
> do
>    echo $i
>    i=$(($i+1))
> done
1
2
3
...
29
30
```

Note que para podermos fazer o incremento da variável **i**, foi utilizado um recurso introduzido pelo Korn Shell, que é o *shell* padrão dos UNIX atuais. A forma **$((expressão))** permite uma operação aritmética com variáveis e constantes numéricas.

Uma outra estrutura de repetição simimar ao **while**, mas com o teste de condição invertido, é:

```
until condicao
do
   comandos
done
```

A única diferença para o **while** é que o **until** executa enquanto a condição é falsa, ou seja, até que a condição seja verdadeira.

A última estrutura de controle, e que é utilizada por exemplo em *scripts* de *boot* estilo System V, que aceitam as opções **start** ou **stop** (vide /etc/rc.d/init.d/smb), é o **case**:

```
case variavel in
   padrao [ | padrão ] ...) comandos;;
   padrao [ | padrão ] ...) comandos;;
```

```
esac
```

Vejamos um exemplo:

```
#!/bin/sh

echo "Agora jah eh noite? Responda sim ou nao"
read noite

case "$noite" in
   "sim" | "Sim" | "SIM") echo "Boa noite!";;
   "nao" | "Nao" | "NAO") echo "Eh hora de trabalhar!";;
   "sei la"              ) echo "Olhe pela janela, por favor.";;
   *                     ) echo "Acho que voce nao entendeu a pergunta";;
esac

exit 0
```

Preparando e executando o *script* temos:

```
$ chmod +x testa-noite.sh
$ ./testa-noite
Agora jah eh noite? Responda sim ou nao
de tarde
Acho que voce nao entendeu a pergunta
```

3.4 Obtendo Ajuda

Muitas outras características estão presentes no *shell*. Muitas vezes o programador fica amarrado aos próprios utilitários, que são importantes para a construção de *scripts* funcionais.

Para ajudar na sintaxe dos comandos, o *shell* possui um comando de ajuda interno. Este comando, chamado **help**, dá a lista de comandos disponíveis, ou a correta sintaxe de um comando específico solicitado.

```
help [ comando ]
```

Por exemplo, obtendo ajuda sobre o comando **while**:

```
$ help while
while: while COMMANDS; do COMMANDS; done
    Expand and execute COMMANDS as long as the final command in the
    `while' COMMANDS has an exit status of zero.
```

Outra fonte importantíssima de informação é a página de manual do próprio *shell*. Muitas coisas deixadas de fora neste texto podem ser obtidas pelo **man**.