Dave Hamilton - www.sxc.hu

Curso de Shell Script

Papo de botequim IV

O garçon já perdeu a conta das cervejas, e o assunto não acaba. Desta vez vamos aprender a testar os mais variados tipos de condições, para podermos controlar a execução de nosso programa de acordo com a entrada fornecida pelo usuário. POR JULIO CEZAR NEVES

E aí cara, tentou fazer o exercício que te pedi em nosso último encontro?

– Claro que sim! Em programação, se você não treinar não aprende. Você me pediu um script para informar se um determinado usuário, cujo nome será passado como parâmetro para o script, está "logado" (arghh!) ou não. Fiz o seguinte:

```
$ cat logado

#!/bin/bash
# Pesquisa se um usuário está
# logado ou não

if who | grep $1
then
    echo $1 está logado
else
    echo $1 não está no pedaço
fi
```

– Calma rapaz! Já vi que você chegou cheio de tesão. Primeiro vamos pedir os nossos chopes de praxe e depois vamos ao Shell. Chico, traz dois chopes, um sem colarinho!

– Aaah! Agora que já molhamos os nossos bicos, vamos dar uma olhada nos resultados do seu programa:

```
$ logado jneves
jneves   pts/0        Oct 18 ➥
12:02 (10.2.4.144)
jneves está logado
```

Realmente funcionou. Passei meu nome de usuário como parâmetro e ele disse que eu estava logado, porém ele imprimiu uma linha extra, que eu não pedi, que é a saída do comando *who*. Para evitar que isso aconteça, é só mandá-la para o buraco negro do mundo UNIX, o */dev/null*. Vejamos então como ficaria:

```
$ cat logado
#!/bin/bash
# Pesquisa se uma pessoa está
# logada ou não (versão 2)
if who | grep $1 > /dev/null
then
    echo $1 está logado
else
    echo $1 não está no pedaço
fi
```

Agora vamos aos testes:

```
$ logado jneves
jneves está logado
$ logado chico
chico não está no pedaço
```

Ah, agora sim! Lembre-se dessa pegadinha: a maior parte dos comandos tem uma saída padrão e uma saída de erros (o *grep* é uma das poucas exceções: ele não exibe uma mensagem de erro quando não acha uma cadeia de caracteres) e devemos redirecioná-las para o buraco negro quando necessário.

Bem, agora vamos mudar de assunto: na última vez que nos encontramos aqui no botequim, quando já estávamos de goela seca, você me perguntou como se testam condições. Para isso, usamos o comando *test*

Testes

Todos estamos acostumados a usar o *if* para testar condições, e estas são sempre *maior que*, *menor que*, *maior ou igual a*, *menor ou igual a*, *igual a* e

Tabela 1 – Opções do *test* para arquivos

Opção	Verdadeiro se
-e arq	*arq* existe
-s arq	*arq* existe e tem tamanho maior que zero
-f arq	*arq* existe e é um arquivo regular
-d arq	*arq* existe e é um diretório
-r arq	*arq* existe e com direito de leitura
-w arq	*arq* existe e com direito de escrita
-x arq	*arq* existe e com direito de execução

Tabela 2 – Opções do *test* para cadeias de caracteres

Opção	Verdadeiro se:
-z cadeia	Tamanho de cadeia é zero
-n cadeia	Tamanho de cadeia é maior que zero
cadeia	A cadeia cadeia tem tamanho maior que zero
c1 = c2	Cadeia *c1* e *c2* são idênticas

diferente de. Para testar condições em Shell Script usamos o comando *test*, só que ele é muito mais poderoso do que aquilo com que estamos acostumados. Primeiramente, veja na Tabela 1 as principais opções (existem muitas outras) para testar arquivos em disco e na Tabela 2 as principais opções para teste de cadeias de caracteres.

Tabela 3 – Opções do *test* para números

Opção	Verdadeiro se	Significado
n1 -eq n2	*n1* e *n2* são iguais	equal
n1 -ne n2	*n1* e *n2* não são iguais	not equal
n1 -gt n2	*n1* é maior que *n2*	greater than
n1 -ge n2	*n1* é maior ou igual a *n2*	greater or equal
n1 -lt n2	*n1* é menor que *n2*	less than
n1 -le n2	*n1* é menor ou igual a *n2*	less or equal

Pensa que acabou? Engano seu! Agora é hora de algo mais familiar, as famosas comparações com valores numéricos. Veja a Tabela 3, e some às opções já apresentadas os operadores da Tabela 4.

Ufa! Como você viu, tem coisa pra chuchu, e o nosso *if* é muito mais poderoso que o dos outros. Vamos ver em uns exemplos como isso tudo funciona. Testamos a existência de um diretório:

```
if  test -d lmb
then
    cd lmb
else
    mkdir lmb
    cd lmb
fi
```

Tabela 4

Operador	Finalidade
Parênteses *()*	o
Exclamação *!*	o
-a	o
-o	o

No exemplo, testei a existência do diretório *lmb*. Se não existisse (*else*), ele seria criado. Já sei, você vai criticar a minha lógica dizendo que o script não está otimizado. Eu sei, mas queria que você o entendesse assim, para então poder usar o "ponto-de-espantação" (!) como um negador do test. Veja só:

```
if  test ! -d lmb
then
    mkdir lmb
fi
cd lmb
```

Desta forma o diretório lmb seria criado somente se ele ainda não existisse, e esta negativa deve-se ao ponto de exclamação (*!*) precedendo a opção *-d*. Ao fim da execução desse fragmento de script, com certeza o programa estaria dentro do diretório *lmb*. Vamos ver dois exemplos para entender a diferença na comparação entre números e entre cadeias de caracteres.

```
cad1=1
cad2=01
if  test $cad1 = $cad2
then
    echo As variáveis são iguais.
else
    echo As variáveis são diferentes.
fi
```

Executando o fragmento de programa acima, teremos como resultado:

```
As variáveis são diferentes.
```

Vamos agora alterá-lo um pouco para que a comparação seja numérica:

```
cad1=1
cad2=01
if  test $cad1 -eq $cad2
then
    echo As variáveis são iguais.
else
    echo As variáveis são diferentes.
fi
```

E vamos executá-lo novamente:

```
As variáveis são iguais.
```

Como você viu, nas duas execuções obtive resultados diferentes, porque a cadeia de caracteres "01" é realmente diferente de "1". Porém, a coisa muda de figura quando as variáveis são testadas numericamente, já que o número 1 é igual ao número 01.

Para mostrar o uso dos conectores *-o* (ou) e *-a* (e), veja um exemplo "animal", programado direto no prompt do Bash. Me desculpem os zoólogos, mas eu não entendo nada de reino, filo, classe, ordem, família, gênero, espécie e outras coisas do tipo, desta forma o que estou chamando de família ou de gênero tem grande chance de estar total e completamente incorreto:

```
$ Familia=felinae
$ Genero=gato
$ if  test $Familia = canidea ➥
-a $Genero = lobo -o $Familia = ➥
felina -a $Genero = leão
> then
>     echo Cuidado
> else
>     echo Pode passar a mão
> fi
Pode passar a mão
```

Neste exemplo, caso o animal fosse da família canídea e (*-a*) do gênero lobo, ou (*-o*) da familia felina e (*-a*) do gênero leão, seria dado um alerta, caso contrário a mensagem seria de incentivo.

Atenção: Os sinais de maior (>) no início das linhas internas ao if são os prompts de continuação (que estão definidos na variável $PS2). Quando o shell identifica que um comando continuará na linha seguinte, automaticamente ele coloca este caractere, até que o comando seja encerrado.

Vamos mudar o exemplo para ver se o programa continua funcionando:

```
$ Familia=felino
$ Genero=gato
$ if  test $Familia = felino -o ➥
$Familia = canideo -a $Genero = ➥
onça -o $Genero = lobo
> then
>     echo Cuidado
> else
>     echo Pode passar a mão
> fi
Cuidado
```

Obviamente a operação resultou em erro, porque a opção *-a* tem precedência

sobre a *-o* e, dessa, forma o que foi avaliado primeiro foi a expressão:

```
$Familia = canideo -a $Genero = ➥
onça
```

Que foi avaliada como falsa, retornando o seguinte:

```
$Familia = felino -o FALSO -o ➥
$Genero = lobo
```

Que resolvida resulta em:

```
VERDADEIRO -o FALSO -o FALSO
```

Como agora todos os conectores são *-o*, e para que uma série de expressões conectadas entre si por diversos "ou" lógicos seja verdadeira, basta que uma delas o seja. A expressão final resultou como VERDADEIRO e o *then* foi executado de forma errada. Para que isso volte a funcionar façamos o seguinte:

```
$ if  test \($Familia = felino ➥
-o $Familia = canideo\) -a ➥
\($Genero = onça -o $Genero = ➥
lobo\)
> then
>     echo Cuidado
> else
>     echo Pode passar a mão
> fi
Pode passar a mão
```

Desta forma, com o uso dos parênteses agrupamos as expressões com o conector *-o*, priorizando a execução e resultando em *VERDADEIRO -a FALSO*.

Para que seja *VERDADEIRO* o resultado de duas expressões ligadas pelo conector *-a*, é necessário que ambas sejam verdadeiras, o que não é o caso do exemplo acima. Assim, o resultado final foi *FALSO*, sendo então o else corretamente executado.

Se quisermos escolher um CD que tenha faixas de 2 artistas diferentes, nos sentimos tentados a usar um *if* com o conector *-a*, mas é sempre bom lembrar que o bash nos oferece muitos recursos e isso poderia ser feito de forma muito mais simples com um único comando *grep*, da seguinte forma:

```
$ grep Artista1 musicas | grep ➥
Artista2
```

Da mesma forma, para escolhermos CDs que tenham a participação do *Artista1* e do *Artista2*, não é necessário montar um *if* com o conector *-o*. O egrep também resolve isso para nós. Veja como:

```
$ egrep (Artista1|Artista2) ➥
musicas
```

Ou (nesse caso específico) o próprio grep poderia nos quebrar o galho:

```
$grep Artista[12] musicas
```

No egrep acima, foi usada uma expressão regular, na qual a barra vertical (|) trabalha como um "ou lógico" e os parênteses são usados para limitar a amplitude deste "ou". Já no grep da linha seguinte, a palavra *Artista* deve ser seguida por um dos valores da lista formada pelos colchetes (*[]*), isto é, 1 ou 2.

– Tá legal, eu aceito o argumento, o *if* do shell é muito mais poderoso que os outros caretas - mas, cá entre nós, essa construção de *if test ...* é muito esquisita, é pouco legível.
– É, você tem razão, eu também não gosto disso e acho que ninguém gosta. Acho que foi por isso que o shell incorporou outra sintaxe, que substitui o comando test.

Para isso vamos pegar aquele exemplo para fazer uma troca de diretórios, que era assim:

```
if  test ! -d lmb
then
    mkdir lmb
fi
cd lmb
```

e utilizando a nova sintaxe, vamos fazê-lo assim:

```
if  [ ! -d lmb ]
then
    mkdir lmb
fi
cd lmb
```

Ou seja, o comando test pode ser substituído por um par de colchetes (*[]*), separados por espaços em branco dos argumentos, o que aumentará enormemente a legibilidade, pois o comando if irá ficar com a sintaxe semelhante à das outras linguagens; por isso, esse será o modo como o comando test será usado daqui para a frente.

Se você pensa que acabou, está muito enganado. Preste atenção à "Tabela Verdade" na Tabela 5.

Tabela 5 - Tabela Verdade

Combinação	E	OU
VERDADEIRO-VERDADEIRO	TRUE	TRUE
VERDADEIRO-FALSO	FALSE	TRUE
FALSO-VERDADEIRO	FALSE	TRUE
FALSO-FALSO	FALSE	FALSE

Ou seja, quando o conector é *e* e a primeira condição é verdadeira, o resultado final pode ser verdadeiro ou falso, dependendo da segunda condição; já no conector *ou*, caso a primeira condição seja verdadeira, o resultado sempre será verdadeiro. Se a primeira for falsa, o resultado dependerá da segunda condição.

Ora, os caras que desenvolveram o interpretador não são bobos e estão sempre tentando otimizar ao máximo os algoritmos. Portanto, no caso do conector *e*, a segunda condição não será avaliada, caso a primeira seja falsa, já que o resultado será sempre falso. Já com o *ou*, a segunda será executada somente caso a primeira seja falsa.

Aproveitando-se disso, uma forma abreviada de fazer testes foi criada. O conector *e* foi batizado de *&&* e o *ou* de *||*. Para ver como isso funciona, vamos usá-los como teste no nosso velho exemplo de troca de diretório, que em sua última versão estava assim:

```
if  [ ! -d lmb ]
then
    mkdir lmb
fi
cd lmb
```

O código acima também poderia ser escrito de maneira abreviada:

```
[ ! -d lmb ] && mkdir lmb
cd dir
```

Também podemos retirar a negação (*!*):

```
[ -d lmb ] || mkdir lmb
cd dir
```

Tabela 6

Caractere	Significado
*	Qualquer caractere ocorrendo zero ou mais vezes
?	Qualquer caractere ocorrendo uma vez
[...]	Lista de caracteres
|	"ou" lógico

No primeiro caso, se o primeiro comando (o *test*, que está representado pelos colchetes) for bem sucedido, isto é, se o diretório *lmb* não existir, o comando *mkdir* será executado porque a primeira condição era verdadeira e o conector era *e*.

No exemplo seguinte, testamos se o diretório *lmb* existia (no anterior testamos se ele não existia) e, caso isso fosse verdade, o *mkdir* não seria executado porque o conector era *ou*. Outra forma de escrever o programa:

```
cd lmb || mkdir lmb
```

Nesse caso, se o comando cd fosse mal sucedido, o diretório lmb seria criado mas não seria feita a mudança de diretório para dentro dele. Para executar mais de um comando dessa forma, é necessário fazer um grupamento de comandos, o que se consegue com o uso de chaves ({}). Veja como seria o modo correto:

```
cd lmb ||
    {
    mkdir lmb
    cd lmb
    }
```

Ainda não está legal porque, caso o diretório não exista, o *cd* exibirá uma mensagem de erro. Veja o modo certo:

```
cd lmb 2> /dev/null ||
    {
    mkdir lmb
    cd lmb
    }
```

Como você viu, o comando *if* nos permitiu fazer um *cd* seguro de diversas maneiras. É sempre bom lembrar que o "seguro" a que me refiro diz respeito ao fato de que ao final da execução você sempre estará dentro de *lmb*, desde que tenha permissão para entrar neste diretório, permissão para criar um subdiretório dentro de *../lmb*, que haja espaço em disco suficiente...

Vejamos um exemplo didático: dependendo do valor da variável *$opc* o script deverá executar uma das opções a seguir: inclusão, exclusão, alteração ou encerrar sua execução. Veja como ficaria o código:

```
if  [ $opc -eq 1 ]
then
    inclusao
elif [ $opc -eq 2 ]
then
    exclusao
elif [ $opc -eq 3 ]
then
    alteracao
elif [ $opc -eq 4 ]
then
    exit
else
    echo Digite uma opção entre ➥
1 e 4
fi
```

Quadro 1 - Script bem-educado

```
#!/bin/bash
#  Programa bem educado que
#  dá bom-dia, boa-tarde ou
#  boa-noite conforme a hora
Hora=$(date +%H)
case $Hora in
    0? | 1[01]) echo Bom Dia
                ;;
    1[2-7]    ) echo Boa Tarde
                ;;
    *         ) echo Boa Noite
                ;;
esac
exit
```

Neste exemplo você viu o uso do comando *elif* como um substituto ou forma mais curta de *else if*. Essa é uma sintaxe válida e aceita, mas poderíamos fazer ainda melhor. Para isso usamos o comando *case*, cuja sintaxe mostramos a seguir:

```
case $var in
    padrao1) cmd1
             cmd2
             cmdn ;;
    padrao2) cmd1
             cmd2
             cmdn ;;
    padraon) cmd1
             cmd2
             cmdn ;;
esac
```

Onde a variável *$var* é comparada aos padrões *padrao1, ..., padraon*. Caso um dos padrões corresponda à variável, o bloco de comandos *cmd1, ..., cmdn* correspondente é executado até encontrar um duplo ponto-e-vírgula (*;;*), quando o fluxo do programa será interrompido e desviado para instrução imediatamente após o comando *esac* (que, caso não tenham notado, é *case* ao contrário. Ele indica o fim do bloco de código, da mesma forma que ot comando *fi* indica o fim de um *if*).

Na formação dos padrões, são aceitos os caracteres mostrados na Tabela 6.

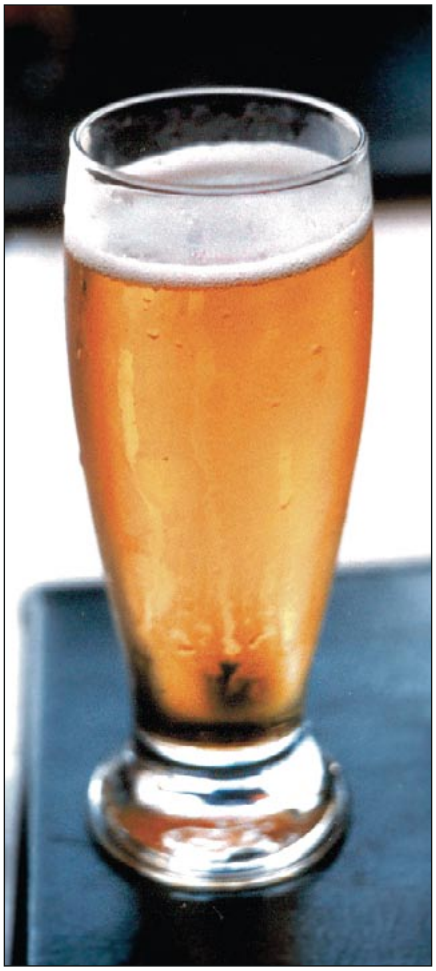
Para mostrar como o código fica melhor, vamos repetir o exemplo anterior, só que desta vez usaremos o *case* em vez do tradicional bloco de código com *if ... elif ... else ... fi*.

```
case $opc in
    1) inclusao ;;
    2) exclusao ;;
    3) alteracao ;;
    4) exit ;;
    *) echo Digite uma opção ➥
entre 1 e 4
esac
```

Como você deve ter percebido, eu usei o asterisco como última opção, isto é, se o asterisco atende a qualquer coisa, então servirá para qualquer coisa que não esteja no intervalo de 1 a 4. Outra coisa a ser notada é que o duplo ponto-e-vírgula não é necessário antes do *esac*.

Vamos agora fazer um script mais radical. Ele te dará bom dia, boa tarde ou boa noite dependendo da hora em que for executado, mas primeiramente veja estes comandos:

```
$ date
Tue Nov  9 19:37:30 BRST 2004
$ date +%H
19
```



O comando *date* informa a data completa do sistema e tem diversas opções de mascaramento do resultado. Neste comando, a formatação começa com um sinal de mais (+) e os caracteres de formatação vêm após um sinal de percentagem (%), assim o *%H* significa a hora do sistema. Dito isso, veja o exemplo no Quadro 1.

Peguei pesado, né? Que nada, vamos esmiuçar a resolução:

0? | 1[01] – Zero seguido de qualquer coisa (*?*), ou (*|*) um seguido de zero ou um (*[01]*), ou seja, esta linha "casa" com *01, 02, ... 09, 10 e 11*;
1[2-7] – Significa um seguido da lista de caracteres entre dois e sete, ou seja, esta linha pega *12, 13, ... 17*;
*** –** Significa tudo o que não casou com nenhum dos padrões anteriores.

– Cara, até agora eu falei muito e bebi pouco. Agora eu vou te passar um exercício para você fazer em casa e me dar a resposta da próxima vez em que nos encontrarmos aqui no botequim, tá legal?
– Beleza!
– É o seguinte: faça um programa que receba como parâmetro o nome de um arquivo e que quando executado salve esse arquivo com o nome original seguido de um til (~) e abra esse arquivo dentro do *vi* para ser editado. Isso é para ter sempre uma cópia de backup do arquivo caso alguém faça alterações indevidas. Obviamente, você fará as críticas necessárias, como verificar se foi passado um parâmetro, se o arquivo indicado existe... Enfim, o que te der na telha e você achar que deva constar do script. Deu pra entender?
– Hum, hum...
– Chico, traz mais um, sem colarinho! ■

SOBRE O AUTOR

Julio Cezar Neves é Analista de Suporte de Sistemas desde 1969 e trabalha com Unix desde 1980, quando participou do desenvolvimento do SOX, um sistema operacional similar ao Unix produzido pela Cobra computadores. Pode ser contatado no e-mail julio.neves@gmail.com