



Dave Hamilton - www.sx.chu

Curso de Shell Script

Papo de Botequim V

Blocos de código e laços (ou *loops*, como preferem alguns)

são o tema do mês em mais uma lição de nosso curso de Shell

Script. Garçom, salta uma boa redondinha, que tô a fim de refrescar o pensamento! **POR JULIO CEZAR NEVES**

Fala cara! E as idéias estão em ordem? Já fundiu a cuca ou você ainda agüenta mais Shell?

- Güento! Tô gostando muito! Gostei tanto que até caprichei no exercício que você passou. Lembra que você me pediu para fazer um programa que recebe como parâmetro o nome de um arquivo e que quando executado salva esse arquivo com o nome original seguido de um til (~) e o abre dentro do vi?
- Claro que lembro, me mostre e explique como você fez.
- Beleza, dá uma olhada no quadro 1
- É, beleza! Mas me diz uma coisa: por que você terminou o programa com um *exit 0*?
- Eu descobri que o número após o *exit* indica o código de retorno do programa (o \$?, lembra?) e assim, como a execução foi bem sucedida, ele encerra com o \$?=0. Porém, se você observar, verá que caso o programa não tenha recebido o nome do arquivo ou caso o operador não tenha permissão de gravação nesse arquivo, o código de retorno (\$?) seria diferente do zero.
- Grande garoto, aprendeu legal, mas é bom deixar claro que *exit 0*, simplesmente *exit* ou não colocar *exit* produzem igualmente um código de retorno (\$?) igual a zero. Agora vamos falar sobre as instruções de loop ou laço, mas antes vou passar o conceito de bloco de código.

Até agora já vimos alguns blocos de código, como quando te mostrei um exemplo para fazer um *cd* para dentro de um diretório:

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

O fragmento contido entre as duas chaves ({}) forma um bloco de código. Também nesse exercício que acabamos de ver, em que salvamos o arquivo antes de editá-lo, existem vários blocos de código compreendidos entre os comandos *then* e *fi* do *if*. Um bloco de código também pode estar dentro de um *case* ou entre um *do* e um *done*.

- Peraí, Julio, que *do* e *done* são esses? Não me lembro de você ter falado nisso, e olha que estou prestando muita atenção...
- Pois é, ainda não tinha falado porque não havia chegado a hora certa.

Todas as instruções de loop ou laço executam os comandos do bloco compreendidos entre um *do* e um *done*. As instruções de loop ou laço são *for*, *while* e *until*, que serão explicadas uma a uma a partir de hoje.

O comando *For*

Se você está habituado a programar, certamente já conhece o comando *for*, mas o que você não sabe é que o *for*,

Quadro 1: *vira.sh*

```
$ cat vira.sh
#!/bin/bash
#
# vira - vi resguardando
# arquivo anterior
# Verifica se algum parâmetro foi
# passado
if [ "$#" -ne 1 ]
then
  echo "Erro -> Uso: $0 >
<arquivo>"
  exit 1
fi
Arq=$1
# Caso o arquivo não exista, não
# há cópia a ser salva
if [ ! -f "$Arq" ]
then
  vi $Arq
  exit 0
fi
# Se eu não puder alterar o
#arquivo, vou usar o vi para que?
if [ ! -w "$Arq" ]
then
  echo "Você não tem permissão >
de escrita em $Arq"
  exit 2
fi
# Já que está tudo OK, vou
# salvar a cópia e chamar o vi
cp -f $Arq $Arq~
vi $Arq
exit 0
```

que é uma instrução intrínseca do Shell (isso significa que o código fonte do comando faz parte do código fonte do Shell, ou seja, em bom programês é um *built-in*), é muito mais poderoso que os seus correlatos das outras linguagens.

Vamos entender a sua sintaxe, primeiro em português e, depois, como funciona pra valer. Olhe só:

```
para var em val1 val2 ... valn
faça
  cmd1
  cmd2
  cmdn
feito
```

Onde a variável *var* assume cada um dos valores da lista *val1 val2 ... valn* e, para cada um desses valores, executa o bloco de comandos formado por *cmd1*, *cmd2* e *cmdn*. Agora que já vimos o significado da instrução em português, vejamos a sintaxe correta:

```
for var in val1 val2 ... valn
do
  cmd1
  cmd2
  cmdn
done
```

Vamos aos exemplos, para entender direito o funcionamento deste comando. Vamos escrever um script para listar todos os arquivos do diretório, separados por dois-pontos, mas antes veja isso:

```
$ echo *
ArqDoDOS.txt1 confuso incusu
logado musexc musicas musinc
muslist
```

Isto é, o Shell viu o asterisco (*), expandiu-o com o nome de todos os arquivos do diretório e o comando *echo* jogou-os para a tela separados por espaços em branco. Visto isso, vamos resolver o problema a que nos propusemos:

```
$ cat testefor1
#!/bin/bash
# 1o. Programa didático para
# entender o for
for Arq in *
do
  echo -n $Arq:
done
```

Então vamos executá-lo:

```
$ testefor1
ArqDoDOS.txt1:confuso:incusu:
logado:musexc:musicas:musinc:
muslist:$
```

Como você viu, o Shell transformou o asterisco (que odeia ser chamado de *asterístico*) em uma lista de arquivos separados por espaços em branco. Quando o *for* viu aquela lista, disse: “Opa, listas separadas por espaços é comigo mesmo!”

O bloco de comandos a ser executado era somente o *echo*, que com a opção *-n* listou a variável *\$Arq* seguida de dois-pontos (:), sem saltar a linha. O cifrão (\$) do final da linha da execução é o prompt, que permaneceu na mesma linha também em função da opção *-n*. Outro exemplo simples (por enquanto):

```
$ cat testefor2
#!/bin/bash
# 2o. Programa didático para
# entender o for
for Palavra in Linux Magazine
do Brasil
do
  echo $Palavra
done
```

E executando temos:

```
$ testefor2
Linux
Magazine
do
Brasil
```

Como você viu, esse exemplo é tão bobo e simples como o anterior, mas serve para mostrar o comportamento básico do *for*. Veja só a força do comando: ainda estamos na primeira possibilidade de sintaxe e já estou mostrando novas formas de usá-lo. Lá atrás eu havia falado que o *for* usava listas separadas por espaços em branco, mas isso é uma meia-verdade, só para facilitar a compreensão. Na verdade, as listas não são obrigatoriamente separadas por espaços. Mas antes de prosseguir, preciso te mostrar como se comporta uma variável do sistema chamada de IFS, ou *Inter Field Separator* Veja no exemplo a seguir seu conteúdo:

```
$ echo "$IFS" | od -h
0000000 0920 0a0a
0000004
```

Isto é, mandei a variável (protegida da interpretação do Shell pelas aspas) para um dump hexadecimal (*od -h*). O resultado pode ser interpretado com a tabela abaixo:

Tabela 1: Resultado do *od -h*

Valor Hexadecimal	Significado
09	<TAB>
20	<ESPAÇO>
0a	<ENTER>

O último *0a* foi proveniente do <ENTER> dado ao final do comando. Para melhorar a explicação, vamos ver isso de outra forma:

```
$ echo "::$IFS:" | cat -vet
: ^I$
: $
```

No comando *cat*, a opção *-e* representa o <ENTER> como um cifrão (\$) e a opção *-t* representa o <TAB> como um ^I. Usei os dois-pontos (:) para mostrar o início e o fim do *echo*. E dessa forma, pudemos notar que os três caracteres estão presentes naquela variável.

Agora veja você: traduzindo, IFS significa *separador entre campos*. Uma vez entendido isso, eu posso afirmar que o comando *for* não usa apenas listas separadas por espaços em branco, mas sim pelo conteúdo da variável *IFS*, cujo valor padrão são os caracteres que acabamos de ver. Para comprovarmos isso, vamos continuar mexendo em nossa CDTEca, escrevendo um script que recebe o nome do artista como parâmetro e lista as músicas que ele toca. Mas primeiramente vamos ver como está o nosso arquivo *musicas*:

```
$ cat musicas
album 1^Artista1~Musical:
Artista2~Musica2
album 2^Artista3~Musica3:
Artista4~Musica4
album 3^Artista5~Musica5:
Artista6~Musica6
album 4^Artista7~Musica7:
Artista1~Musica3
```

```
album 5^Artista9~Musica9:~
Artista10~Musica10
```

Em cima desse “leiaute” desenvolvemos o script a seguir:

```
$ cat listartista
#!/bin/bash
# Dado um artista, mostra as
# suas músicas
if [ $# -ne 1 ]
then
    echo Você deveria ter ~
    passado um parâmetro
    exit 1
fi
IFS="
:"
for ArtMus in $(cut -f2 -d^ ~
musicas)
do
    echo "$ArtMus" | grep $1 && ~
    echo $ArtMus | cut -f2 -d~
done
```

O script, como sempre, começa testando se os parâmetros foram passados corretamente, em seguida o IFS foi configurado para <ENTER> e dois-pontos (:) (como demonstram as aspas em linhas diferentes), porque é ele quem separa os blocos *Artista~Musica*. Desta forma, a variável *\$ArtMus* irá receber cada um desses blocos do arquivo (repare que o *for* já recebe os registros sem o álbum em virtude do *cut* na sua linha). Caso encontre o parâmetro (*\$1*) no bloco, o segundo *cut* listará somente o nome da música. Vamos executar o programa:

```
$ listartista Artista1
Artista1~Musical
Musica1
Artista1~Musica3
Musica3
Artista10~Musica10
Musica10
```

Êpa! Aconteceram duas coisas indesejáveis: os blocos também foram listados, e a *Musica10* idem. Além do mais, o nosso arquivo de músicas está muito simples: na vida real, tanto a música quanto o artista têm mais de um nome. Suponha que o artista fosse uma dupla sertaneja chamada *Perereca & Peteleca* (não gosto nem de dar a idéia com receio que isso se torne realidade). Nesse caso,

o *\$1* seria *Perereca* e o resto desse lindo nome seria ignorado na pesquisa.

Para que isso não ocorra, eu deveria passar o nome do artista entre aspas (") ou trocar *\$1* por *\$** (que representa todos os parâmetros passados), que é a melhor solução, mas nesse caso eu teria que modificar a crítica dos parâmetros e o *grep*. A nova versão não seria se eu passei um parâmetro, mas sim se passei pelo menos um parâmetro. Quanto ao *grep*, veja só o que aconteceria após a substituição do *\$** pelos parâmetros:

```
echo "$ArtMus" | grep perereca ~
& peteleca
```

Isso gera um erro. O correto é:

```
echo "$ArtMus" | grep -i ~
"perereca & peteleca"
```

Aqui adicionamos a opção *-i* para que a pesquisa ignorasse maiúsculas e minúsculas. As aspas foram inseridas para que o nome do artista fosse visto como uma só cadeia de caracteres.

Falta consertar o erro dele ter listado o *Artista10*. O melhor é dizer ao *grep* que a cadeia de caracteres está no início (^) de *\$ArtMus* e que logo após vem um til (~). É preciso redirecionar a saída do *grep* para */dev/null* para que os blocos não sejam listados. Veja a nova cara do programa:

```
$ cat listartista
#!/bin/bash
# Dado um artista, mostra as
# suas musicas
# Versao 2
if [ $# -eq 0 ]
then
    echo Voce deveria ter ~
    passado pelo menos um parametro
    exit 1
fi
IFS="
:"
for ArtMus in $(cut -f2 -d^ ~
musicas)
do
    echo "$ArtMus" | grep -i ~
"^$*~" > /dev/null && echo ~
$ArtMus | cut -f2 -d~
done
```

O resultado é:

```
$ listartista Artista1
Musica1
Musica3
```

Veja uma segunda sintaxe para o *for*:

```
for var
do
    cmd1
    cmd2
    cmdn
done
```

Ué, sem o *in*, como ele vai saber que valor assumir? Pois é, né? Esta construção, à primeira vista, parece esquisita, mas é bastante simples. Neste caso, *var* assumirá um a um cada parâmetro passado para o programa. Como exemplo para entender melhor, vamos fazer um script que receba como parâmetro um monte de músicas e liste seus autores:

```
$ cat listamusica
#!/bin/bash
# Recebe parte dos nomes de
# músicas como parâmetro e
# lista os intérpretes. Se o
# nome for composto, deve
# ser passado entre aspas.
# ex. "Eu não sou cachorro não"
# "Churrasquinho de Mãe"
#
if [ $# -eq 0 ]
then
    echo Uso: $0 musical ~
[musica2] ... [musicam]
    exit 1
fi
IFS="
:"
for Musica
do
    echo $Musica
    Str=$(grep -i "$Musica" ~
musicas) ||
    {
        echo " Não ~
encontrada"
        continue
    }
    for ArtMus in $(echo "$Str" ~
| cut -f2 -d^ )
    do
        echo " $ArtMus" | ~
grep -i "$Musica" | cut -f1 -d~
    done
done
```

Da mesma forma que os outros, começamos o exercício com uma crítica sobre os parâmetros recebidos, em seguida fizemos um *for* em que a variável *\$Musica* receberá cada um dos parâmetros passados, colocando em *\$Str* todos os álbuns que contêm as músicas desejadas. Em seguida, o outro *for* pega cada bloco *Artista~Musica* nos registros que estão em *\$Str* e lista cada artista que toca aquela música. Vamos executar o programa para ver se funciona mesmo:

```
$ listamusica musica3 Musica4
"Egüinha Pocotó"
musica3
  Artista3
  Artista1
Musica4
  Artista4
Egüinha Pocotó
  Não encontrada
```

A listagem ficou feinha porque ainda não sabemos formatar a saída; mas qualquer dia desses, quando você souber posicionar o cursor, trabalhar com cores etc., faremos esse programa novamente usando todas essas perfumarias.

A esta altura dos acontecimentos, você deve estar se perguntando: "E aquele *for* tradicional das outras linguagens em que ele sai contando a partir de um número, com um determinado incremento, até alcançar uma condição?". E é aí que eu te respondo: "Eu não te disse que o nosso *for* é mais porreta que o dos outros?" Para fazer isso, existem duas formas. Com a primeira sintaxe que vimos, como no exemplo:

```
for i in $(seq 9)
do
  echo -n "$i "
done
1 2 3 4 5 6 7 8 9
```

A variável *i* assumiu os valores inteiros entre 1 a 9 gerados pelo comando *seq* e a opção *-n* do *echo* foi usada para não saltar uma linha a cada número listado. Ainda usando o *for* com *seq*:

```
for i in $(seq 4 9)
do
  echo -n "$i "
done
4 5 6 7 8 9
```

Ou na forma mais completa do *seq*:

```
for i in $(seq 0 3 9)
do
  echo -n "$i "
done
0 3 6 9
```

A outra forma de fazer isso é com uma sintaxe muito semelhante ao *for* da linguagem C, como vemos a seguir:

```
for ((var=ini; cond; incr))
do
  cmd1
  cmd2
  cmdn
done
```

Onde *var=ini* significa que a variável *var* começará de um valor inicial *ini*; *cond* significa que o loop ou laço *for* será executado enquanto *var* não atingir a condição *cond* e *incr* significa o incremento que a variável *var* sofrerá a cada passada do loop. Vamos aos exemplos:

```
for ((i=1; i<=9; i++))
do
  echo -n "$i "
done
1 2 3 4 5 6 7 8 9
```

A variável *i* partiu do valor inicial 1, o bloco de código (aqui somente o *echo*) será executado enquanto *i* for menor ou igual (*<=*) a 9 e o incremento de *i* será de 1 a cada passada do loop.

Repare que no *for* propriamente dito (e não no bloco de código) não coloquei um cifrão (\$) antes do *i* e a notação para incrementar (*i++*) é diferente do que vimos até agora. O uso de parênteses duplos (assim como o comando *let*) chama o interpretador aritmético do Shell, que é mais tolerante.

Só para mostrar como o *let* funciona e a versatilidade do *for*, vamos fazer a mesma coisa, mas omitindo a última parte do escopo do *for*, passando-a para o bloco de código:

```
for ((; i<=9;))
do
  let i++
  echo -n "$i "
done
1 2 3 4 5 6 7 8 9
```

Repare que o incremento saiu do corpo do *for* e passou para o bloco de código; repare também que, quando usei o *let*, não foi necessário inicializar a variável *\$i*. Veja só os comandos a seguir, digitados diretamente no prompt, para demonstrar o que acabo de falar:

```
$ echo $j
$ let j++
$ echo $j
1
```

Ou seja, a variável *\$j* sequer existia e no primeiro *let* assumiu o valor 0 (zero) para, após o incremento, ter o valor 1. Veja só como as coisas ficam simples:

```
for arq in *
do
  let i++
  echo "$i -> $Arq"
done
1 -> ArqDoDOS.txt1
2 -> confuso
3 -> incusu
4 -> listamusica
5 -> listartista
6 -> logado
7 -> musexc
8 -> musicas
9 -> musinc
10 -> muslist
11 -> testefor1
12 -> testefor2
```

- Pois é amigo, tenho certeza que você já tomou um xarope do comando *for*. Por hoje chega, na próxima vez em que nos encontrarmos falaremos sobre outras instruções de loop, mas eu gostaria que até lá você fizesse um pequeno script para contar a quantidade de palavras de um arquivo texto, cujo nome seria recebido como parâmetro. Essa contagem **tem** que ser feita com o comando *for*, para se habituar ao seu uso. Não vale usar o *wc -w*. Aê Chico! Traz a saideira! ■

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