Curso de Shell Script

Papo de Botequim

Você não agüenta mais aquele seu amigo usuário de Linux enchendo o seu saco com aquela história de que o sistema é fantástico e o Shell é uma ferramenta maravilhosa? A partir desta edição vai ficar mais fácil entender o porquê deste entusiasmo...

POR JULIO CEZAR NEVES

Diálogo entreouvido em uma mesa de um botequim, entre um usuário de Linux e um empurrador de mouse:

- Quem é o Bash?
- É o filho caçula da família Shell.
- Pô cara! Estás a fim de me deixar maluco? Eu tinha uma dúvida e você me deixa com duas!
- Não, maluco você já é há muito tempo: desde que decidiu usar aquele sistema operacional que você precisa reiniciar dez vezes por dia e ainda por cima não tem domínio nenhum sobre o que esta acontecendo no seu computador. Mas deixa isso prá lá, pois vou te explicar o que é Shell e os componentes de sua família e ao final da nossa conversa você dirá: "Meu Deus do Shell! Porque eu não optei pelo Linux antes?".

O ambiente Linux

Para você entender o que é e como funciona o Shell, primeiro vou te mostrar como funciona o ambiente em camadas do Linux. Dê uma olhada no gráfico mostrado na Figura 1.

Neste gráfico podemos ver que a camada de hardware é a mais profunda e é formada pelos componentes físicos do seu computador. Em torno dela, vem a camada do kernel que é o cerne do Linux, seu núcleo, e é quem põe o hardware para funcionar, fazendo seu gerenciamento e controle. Os programas e comandos que envolvem o kernel, dele se utilizam para realizar as tarefas para que foram desenvolvidos. Fechando tudo isso vem o Shell, que leva este nome porque, em inglês, Shell significa concha, carapaça, isto é, fica entre o usuário e o sistema operacional, de forma que tudo que interage com o sistema operacional, tem que passar pelo seu crivo.

O ambiente Shell

Bom já que para chegar ao núcleo do Linux, no seu kernel que é o que interessa a todo aplicativo, é necessária a filtragem do Shell, vamos entender como ele funciona de forma a tirar o máximo proveito das inúmeras facilidades que ele nos oferece.

O Linux, por definição, é um sistema multiusuário – não podemos nunca nos esquecer disto – e para permitir o acesso de determinados usuários e barrar a entrada de outros, existe um arquivo chamado */etc/passwd*, que além de fornecer dados para esta função de "leão-de-chácara" do Linux, também provê informações para o início de uma sessão (ou "login", para os íntimos) daqueles que passaram por esta primeira barreira. O último campo de seus registros informa ao sistema qual é o Shell que a pessoa vai receber ao iniciar sua sessão.

Lembra que eu te falei de Shell, família, irmão? Pois é, vamos começar a entender isto: o Shell é a conceituação de concha envolvendo o sistema operacional propriamente dito, é o nome genérico para tratar os filhos desta idéia que, ao longo dos muitos anos de exis-

Quadro 1: Uma rapidinha nos principais sabores de Shell

Bourne Shell (sh): Desenvolvido por Stephen Bourne do Bell Labs (da AT&T, onde também foi desenvolvido o Unix), foi durante muitos anos o Shell padrão do sistema operacional Unix. É também chamado de Standard Shell por ter sido durante vários anos o único, e é até hoje o mais utilizado. Foi portado para praticamente todos os ambientes Unix e distribuições Linux.

Korn Shell (ksh): Desenvolvido por David Korn, também do Bell Labs, é um supercon-junto do sh, isto é, possui todas as facilidades do sh e a elas agregou muitas outras. A compatibilidade total com o sh vem trazendo muitos usuários e programadores de Shell para este ambiente.

Boune Again Shell (bash): Desenvolvido inicialmente por Brian Fox e Chet Ramey, este é o Shell do projeto GNU. O número de seus adeptos é o que mais cresce em todo o mundo, seja por que ele é o Shell padrão do Linux, seja por sua grande diversidade de comandos, que incorpora inclusive diversos comandos característicos do C Shell.

C Shell (csh): Desenvolvido por Bill Joy, da Universidade de Berkley, é o Shell mais utilizado em ambientes BSD. Foi ele quem introduziu o histórico de comandos. A estruturação de seus comandos é bem similar à da linguagem C. Seu grande pecado foi ignorar a compatibilidade com o sh, partindo por um caminho próprio.

Além destes Shells existem outros, mas irei falar somente sobre os três primeiros, tratando-os genericamente por Shell e assinalando as especificidades de cada um.

tência do sistema operacional Unix, foram aparecendo. Atualmente existem diversos sabores de Shell (veja Quadro 1 na página anterior).

Como funciona o Shell

O Shell é o primeiro programa que você ganha ao iniciar sua sessão (se quisermos assassinar a língua portuguesa podemos também dizer "ao se logar") no Linux. É ele quem vai resolver um monte de coisas de forma a não onerar o kernel com tarefas repetitivas, poupando-o para tratar assuntos mais nobres. Como cada usuário possui o seu próprio Shell interpondo-se entre ele e o Linux, é o Shell quem interpreta os comandos digitados e examina as suas sintaxes, passando-os esmiuçados para execução.

- Êpa! Esse negócio de interpretar comando não tem nada a ver com interpretador não, né?
- Tem sim: na verdade o Shell é um interpretador que traz consigo uma poderosa linguagem com comandos de alto nível, que permite construção de loops, de tomadas de decisão e de armazenamento de valores em variáveis, como vou te mostrar.
- Vou explicar as principais tarefas que o Shell cumpre, na sua ordem de execução. Preste atenção, porque esta ordem é fundamental para o entendimento do resto do nosso bate papo.

Análise da linha de comando

Neste exame o Shell identifica os caracteres especiais (reservados) que têm significado para a interpretação da linha e logo em seguida verifica se a linha passada é um comando ou uma atribuição de valores, que são os ítens que vou descrever a seguir.

Comando

Quando um comando é digitado no "prompt" (ou linha de comando) do Linux, ele é dividido em partes, separadas por espaços em branco: a primeira parte é o nome do programa, cuja existência será verificada; em seguida, nesta ordem, vêm as opções/parâmetros, redirecionamentos e variáveis.

Quando o programa identificado existe, o Shell verifica as permissões dos arquivos envolvidos (inclusive o próprio programa), e retorna um erro caso o usuário que chamou o programa não esteja autorizado a executar esta tarefa.

Com que Shell eu vou?

Quando digo que o último campo do arquivo */etc/passwd* informa ao sistema qual é o Shell que o usuário vai usar ao se "logar", isto deve ser interpretado ao pé-da-letra. Se este campo do seu registro contém o termo *prog*, ao acessar o sistema o usuário executará o programa *prog*. Ao término da execução, a sessão do usuário se encerra automaticamente. Imagine quanto se pode incrementar a segurança com este simples artifício.

```
$ ls linux
linux
```

Neste exemplo o Shell identificou o ls como um programa e o linux como um parâmetro passado para o programa ls.

Atribuição

Se o Shell encontra dois campos separados por um sinal de igual (=) sem espaços em branco entre eles, ele identifica esta seqüência como uma atribuição.

```
$ valor=1000
```

Neste caso, por não haver espaços em branco (que é um dos caracteres reservados), o Shell identificou uma atribuição e colocou 1000 na variável valor.

Resolução de Redirecionamentos

Após identificar os componentes da linha que você digitou, o Shell parte para a resolução de redirecionamentos.

O Shell tem incorporado ao seu elenco de habilidades o que chamamos de redirecionamento, que pode ser de entrada (*stdin*), de saída (*stdout*) ou dos erros (*stderr*), conforme vou explicar a seguir. Mas antes precisamos falar de...

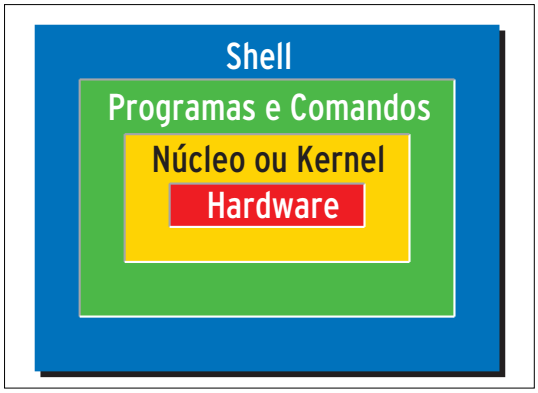


Figura 1: Ambiente em camadas de um sistema Linux

Substituição de Variáveis

Neste ponto, o Shell verifica se as eventuais variáveis (parâmetros começados por $), encontradas no escopo do comando, estão definidas e as substitui por seus valores atuais.

Substituição de Meta-Caracteres

Se algum meta-caracter (ou "coringa", como *, ? ou []*) for encontrado na linha de comando, ele será substituído por seus possíveis valores.

Supondo que o único item no seu diretório corrente cujo nome começa com a letra *n* seja um diretório chamado *nomegrandeprachuchu*, se você fizer:

```
$ cd n*
```

como até aqui quem está manipulando a linha de comando ainda é o Shell e o programa *cd* ainda não foi executado, o Shell expande o *n** para *nomegrandeprachuchu* (a única possibilidade válida) e executa o comando *cd* com sucesso.

Entrega da linha de comando para o kernel

Completadas todas as tarefas anteriores, o Shell monta a linha de comando, já com todas as substituições feitas e chama o kernel para executá-la em um novo Shell (Shell filho), que ganha um número de processo (PID ou Process IDentification) e fica inativo, tirando uma soneca durante a execução do programa. Uma vez encerrado este processo (e o Shell filho), o "Shell pai" recebe novamente o controle e exibe um "prompt", mostrando que está pronto para executar outros comandos.

Cuidado na Atribuição

Jamais faça:

```
$ valor = 1000
bash: valor: not found
```

Neste caso, o Bash achou a palavra valor isolada por espaços e julgou que você estivesse mandando executar um programa chamado *valor*, para o qual estaria passando dois parâmetros: = e *1000*.

Decifrando a Pedra de Roseta

Para tirar aquela sensação que você tem quando vê um script Shell, que mais parece uma sopa de letrinhas ou um conjunto de hieróglifos, vou lhe mostrar os principais caracteres especiais para que você saia por aí como Champollion decifrando a Pedra de Roseta.

Caracteres para remoção do significado.

É isso mesmo, quando não desejamos que o Shell interprete um caractere específico, devemos "escondê-lo" dele. Isso pode ser feito de três maneiras diferentes, cada uma com sua peculiaridade:

- **Apóstrofo (')**: quando o Shell vê uma cadeia de caracteres entre apóstrofos, ele retira os apóstrofos da cadeia e não interpreta seu conteúdo.

```
$ ls linuxm*
linuxmagazine
$ ls 'linuxm*'
bash: linuxm* no such file
or directory
```

No primeiro caso o Shell "expandiu" o asterisco e descobriu o arquivo *linuxmagazine* para listar. No segundo, os apóstrofos inibiram a interpretação do Shell e veio a resposta que não existe o arquivo *linuxm**.

- **Contrabarra ou Barra Invertida (\)**: idêntico aos apóstrofos exceto que a barra invertida inibe a interpretação somente do caractere que a segue.
  Suponha que você, acidentalmente, tenha criado um arquivo chamado * (asterisco) – o que alguns sabores de Unix permitem – e deseja removê-lo. Se você fizesse:

```
$ rm *
```

Você estaria na maior encrenca, pois o *rm* removeria todos os arquivos do diretório corrente. A melhor forma de fazer o serviço é:

```
$ rm \*
```

Desta forma, o Shell não interpreta o asterisco, evitando a sua expansão.
Faça a seguinte experiência científica:

```
$ cd /etc
$ echo '*'
$ echo \*
$ echo *
```

Viu a diferença?

- **Aspas ("):** exatamente iguais ao apóstrofo, exceto que, se a cadeia entre aspas contiver um cifrão ($), uma crase (`), ou uma barra invertida (\), estes caracteres serão interpretados pelo Shell.
  Não precisa se estressar, eu não te dei exemplos do uso das aspas por que você ainda não conhece o cifrão ($) nem a crase (`). Daqui para frente - veremos com muita constância o uso destes caracteres especiais; o mais importante é entender seu significado.

Caracteres de redirecionamento

A maioria dos comandos tem uma entrada, uma saída e pode gerar erros. Esta entrada é chamada Entrada Padrão ou *stdin* e seu dispositivo padrão é o teclado do terminal. Analogamente, a saída do comando é chamada Saída Padrão ou *stdout* e seu dispositivo padrão é a tela do terminal. Para a tela também são enviadas normalmente as mensagens de erro oriundas dos comandos, chamada neste caso de Saída de Erro Padrão ou *stderr*. Veremos agora como alterar este estado de coisas.

Vamos fazer um programa gago. Para isto digite (tecle "Enter" ao final de cada linha – comandos do usuário são ilustrados em negrito):

```
$ cat
E-e-eu sou gago. Vai encarar?
E-e-eu sou gago. Vai encarar?
```

O *cat* é um comando que lista o conteúdo do arquivo especificado para a Saída Padrão (*stdout*). Caso a entrada não seja definida, ele espera os dados da *stdin* (a entrada padrão). Ora como eu não especifiquei a entrada, ele a está esperando pelo teclado (Entrada Padrão) e como também não citei a saída, o que eu teclar irá para a tela (Saída Padrão), criando desta forma – como eu havia proposto – um programa gago. Experimente!

Redirecionamentop da Saída Padrão

Para especificarmos a saída de um programa usamos o símbolo " > " ou o " > > ", seguido do nome do arquivo para o qual se deseja mandar a saída.

Vamos transformar o programa anterior em um "editor de textos":

```
$ cat > Arq
```

O *cat* continua sem ter a entrada especificada, portanto está aguardando que os dados sejam teclados, porém a sua saída está sendo desviada para o arquivo *Arq*. Assim sendo, tudo que esta sendo teclado esta indo para dentro de *Arq*, de forma que fizemos o editor de textos mais curto e ruim do planeta.

Se eu fizer novamente:

```
$ cat > Arq
```

Os dados contidos em *Arq* serão perdidos, já que antes do redirecionamento o Shell criará um *Arq* vazio. Para colocar mais informações no final do arquivo eu deveria ter feito:

```
$ cat >> Arq
```

Redirecionamento da Saída de Erro Padrão

Assim como por padrão o Shell recebe os dados do teclado e envia a saída para a tela, os erros também vão para a tela se você não especificar para onde eles devem ser enviados. Para redirecionar os erros, use *2 > SaidaDeErro*. Note que entre o número *2* e o sinal de maior (>) não existe espaço em branco.

Vamos supor que durante a execução de um script você pode, ou não (dependendo do rumo tomado pela execução do programa), ter criado um arquivo chamado */tmp/seraqueexiste$$*. Como não quer ficar com sujeira no disco rígido, ao final do script você coloca a linha a seguir:

```
rm /tmp/seraqueexiste$$
```

Redirecionamento Perigoso

Como já havia dito, o Shell resolve a linha e depois manda o comando para a execução. Assim, se você redirecionar a saída de um arquivo para ele próprio, primeiramente o Shell "esvazia" este arquivo e depois manda o comando para execução! Desta forma, para sua alegria, você acabou de perder o conteúdo de seu querido arquivo.

Dados ou Erros?

Preste atenção! Não confunda >> com 2>. O primeiro anexa dados ao final de um arquivo, e o segundo redireciona a Saída de Erro Padrão (*stderr*) para um arquivo que está sendo designado. Isto é importante!

Caso o arquivo não existisse seria enviado para a tela uma mensagem de erro. Para que isso não aconteça faça:

```
rm /tmp/seraqueexiste$$ 2> ➥
/dev/null
```

Para que você teste a Saída de Erro Padrão direto no prompt do seu Shell, vou dar mais um exemplo. Faça:

```
$ ls naoexiste
bash: naoexiste no such file ➥
or directory
$ ls naoexiste 2> arquivodeerros
$
$ cat arquivodeerros
bash: naoexiste no such file ➥
or directory
```

Neste exemplo, vimos que quando fizemos um *ls* em *naoexiste*, ganhamos uma mensagem de erro. Após redirecionar a Saída de Erro Padrão para *arquivodeerros* e executar o mesmo comando, recebemos somente o "prompt" na tela. Quando listamos o conteúdo do arquivo para o qual foi redirecionada a Saída de Erro Padrão, vimos que a mensagem de erro tinha sido armazenada nele.

É interessante notar que estes caracteres de redirecionamento são cumulativos, isto é, se no exemplo anterior fizéssemos o seguinte:

```
$ ls naoexiste 2>> ➥
arquivodeerros
```

a mensagem de erro oriunda do *ls* seria anexada ao final de *arquivodeerros*.

Redirecionamento da Entrada Padrão

Para fazermos o redirecionamento da Entrada Padrão usamos o < (menor que). "E pra que serve isso?", você vai me perguntar. Deixa eu dar um exemplo, que você vai entender rapidinho.

Suponha que você queira mandar um mail para o seu chefe. Para o chefe nós caprichamos, né? Então ao invés de sair redigindo o mail direto no "prompt", de forma a tornar impossível a correção de uma frase anterior onde, sem querer, você escreveu um "nós vai", você edita um arquivo com o conteúdo da mensagem e após umas quinze verificações sem constatar nenhum erro, decide enviá-lo e para tal faz:

```
$ mail chefe@chefia.com.br < ➥
arquivocommailparaochefe
```

e o chefe receberá uma mensagem com o conteúdo do *arquivocommailparaochefe*.

Outro tipo de redirecionamento "muito louco" que o Shell permite é o chamado "here document". Ele é representado por < < e serve para indicar ao Shell que o escopo de um comando começa na linha seguinte e termina quando encontra uma linha cujo conteúdo seja unicamente o "label" que segue o sinal < <.

Veja o fragmento de script a seguir, com uma rotina de ftp:

```
ftp -ivn hostremoto << fimftp
user $Usuario $Senha
binary
get arquivoremoto
fimftp
```

neste pedacinho de programa temos um monte de detalhes interessantes:

- As opções usadas para o *ftp* (*-ivn*) servem para ele listar tudo que está acontecendo (opção *-v* de "verbose"), para não ficar perguntando se você tem certeza que deseja transmitir cada arquivo (opção *-i* de "interactive") e finalmente a opção *-n* serve para dizer ao *ftp* para ele não solicitar o usuário e sua senha, pois estes serão informados pela instrução específica (*user*);
- Quando eu usei o < < *fimftp*, estava dizendo o seguinte para o interpretador: "Olha aqui Shell, não se meta em nada a partir deste ponto até encontrar o 'label' *fimftp*. Você não entenderia droga nenhuma, já que são instruções específicas do *ftp*".

  Se fosse só isso seria simples, mas pelo próprio exemplo dá para ver que existem duas variáveis (*$Usuario* e *$Senha*), que o Shell vai resolver antes do redirecionamento. Mas a grande vantagem deste tipo de construção é que ela permite que comandos também sejam interpretados dentro do escopo do "here document", o que, aliás, contraria o que acabei de dizer. Logo a seguir te explico como esse negócio funciona. Agora ainda não dá, estão faltando ferramentas.
- O comando *user* é do repertório de instruções do *ftp* e serve para passar o usuário e a senha que haviam sido lidos em uma rotina anterior a este fragmento de código e colocados respectivamente nas duas variáveis: *$Usuario* e *$Senha*.
- O *binary* é outra instrução do *ftp*, que serve para indicar que a transferência de *arquivoremoto* será feita em modo binário, isto é o conteúdo do arquivo não será inteerpretado para saber se está em ASCII, EBCDIC, …
- O comando *get arquivoremoto* diz ao cliente *ftp* para pegar este arquivo no servidor *hostremoto* e trazê-lo para a nossa máquina local. Se quiséssemos enviar um arquivo, bastaria usar, por exemplo, o comando *put arquivolocal*.

Direito de Posse

O *$$* contém o PID, isto é, o número do seu processo. Como o Linux é multiusuário, é bom anexar sempre o *$$* ao nome dos seus arquivos para não haver problema de propriedade, isto é, caso você batizasse o seu arquivo simplesmente como *seraqueexiste*, a primeira pessoa que o usasse (criando-o então) seria o seu dono e a segunda ganharia um erro quando tentasse gravar algo nele.

Etiquetas Erradas

Um erro comum no uso de labels (como o fimftp do exemplo anterior) é causado pela presença de espaços em branco antes ou após o mesmo. Fique muito atento quanto a isso, por que este tipo de erro costuma dar uma boa surra no programador, até que seja detectado. Lembre-se: um label que se preze tem que ter uma linha inteira só para ele.

Redirecionamento de comandos

Os redirecionamentos de que falamos até agora sempre se referiam a arquivos, isto é, mandavam para arquivo, recebiam de arquivo, simulavam arquivo local, … O que veremos a partir de agora, redireciona a saída de um comando para a entrada de outro. É utilíssimo e, apesar de não ser macaco gordo, sempre quebra os

maiores galhos. Seu nome é "pipe" (que em inglês significa tubo, já que ele canaliza a saída de um comando para a entrada de outro) e sua representação é a | (barra vertical).

```
$ ls | wc -l
21
```

O comando *ls* passou a lista de arquivos para o comando *wc*, que quando está com a opção *-l* conta a quantidade de linhas que recebeu. Desta forma, podemos afirmar categoricamente que no meu diretório existiam 21 arquivos.

```
$ cat /etc/passwd | sort | lp
```

A linha de comandos acima manda a listagem do arquivo */etc/passwd* para a entrada do comando *sort*. Este a classifica e envia para o *lp* que é o gerenciador da fila de impressão.

Caracteres de ambiente

Quando queremos priorizar uma expressão, nós a colocamos entre parênteses, não é? Pois é, por causa da aritmética é normal pensarmos deste jeito. Mas em Shell o que prioriza mesmo são as crases (`) e não os parênteses. Vou dar exemplos para você entender melhor.

Eu quero saber quantos usuários estão "logados" no computador que eu administro. Eu posso fazer:

```
$ who | wc -l
8
```

O comando *who* passa a lista de usuários conectados ao sistema para o comando *wc -l*, que conta quantas linhas recebeu e mostra a resposta na tela. Muito bem, mas ao invés de ter um número oito solto na tela, o que eu quero mesmo é que ele esteja no meio de uma frase. Ora, para mandar frases para a tela eu só preciso usar o comando *echo*; então vamos ver como é que fica:

Buraco Negro

Em Unix existe um arquivo fantasma. Chama-se */dev/null*. Tudo que é enviado para este arquivo some. Assemelha-se a um Buraco Negro. No caso do exemplo, como não me interessava guardar a possível mensagem de erro oriunda do comando *rm*, redirecionei-a para este arquivo.

```
$ echo "Existem who | wc -l ➥
usuarios conectados"
Existem who | wc -l usuarios ➥
conectados
```

Hi! Olha só, não funcionou! É mesmo, não funcionou e não foi por causa das aspas que eu coloquei, mas sim por que eu teria que ter executado o *who | wc -l* antes do echo. Para resolver este problema, tenho que priorizar a segunda parte do comando com o uso de crases:

```
$ echo "Existem `who | wc -l` ➥
usuarios conectados"
Existem      8 usuarios ➥
conectados
```

Para eliminar esse monte de brancos antes do 8 que o *wc -l* produziu, basta retirar as aspas. Assim:

```
$ echo Existem `who | wc -l` ➥
usuarios conectados
Existem 8 usuarios conectados
```

As aspas protegem da interpretação do Shell tudo que está dentro dos seus limites. Como para o Shell basta um espaço em branco como separador, o monte de espaços será trocado por um único após a retirada das aspas.

Outra coisa interessante é o uso do ponto-e-vírgula. Quando estiver no Shell, você deve sempre dar um comando em cada linha. Para agrupar comandos em uma mesma linha, temos que separá-los por ponto-e-vírgula. Então:

```
$ pwd ; cd /etc; pwd ;cd -;pwd
/home/meudir
/etc
/home/meudir
```

Neste exemplo, listei o nome do diretório corrente com o comando *pwd*, mudei para o diretório */etc*, novamente listei o nome do diretório e finalmente voltei para o diretório onde estava anteriormente (*cd -*), listando seu nome. Repare que coloquei o ponto-e-vírgula de todas as formas possíveis, para mostrar que não importa se existem espaços em branco antes ou após este caracter.

Finalmente, vamos ver o caso dos parênteses. No exemplo a seguir, colocamos diversos comandos separados por ponto-e-vírgula entre parênteses:

```
$ (pwd ; cd /etc ; pwd)
/home/meudir
/etc
$ pwd
/home/meudir
```

"Quequeiiisso" minha gente? Eu estava no */home/meudir*, mudei para o */etc*, constatei que estava neste diretório com o *pwd* seguinte e quando o agrupamento de comandos terminou, eu vi que continuava no */etc/meudir*!

Hi! Será que tem coisa do mágico Mandrake por aí? Nada disso. O interessante do uso de parênteses é que eles invocam um novo Shell para executar os comandos que estão em seu interior. Desta forma, fomos realmente para o diretório */etc*, porém após a execução de todos os comandos, o novo Shell que estava no diretório */etc* morreu e retornamos ao Shell anterior que estava em */home/meudir*.

Que tal usar nossos novos conceitos?

```
$ mail suporte@linux.br << FIM
Ola suporte, hoje as `date ➥
"+%hh:mm"` ocorreu novamente ➥
aquele problema que eu havia ➥
reportado por telefone. De ➥
acordo com seu pedido segue a ➥
listagem do diretorio:
`ls -l`
Abracos a todos.
FIM
```

Finalmente agora podemos demonstrar o que conversamos anteriormente sobre "here document". Os comandos entre crases tem prioridade, portanto o Shell os executará antes do redirecionamento do "here document". Quando o suporte receber a mensagem, verá que os comandos *date* e *ls* foram executados antes do comando mail, recebendo então um instantâneo do ambiente no momento de envio do email.

- Garçom, passa a régua! ■

SOBRE O AUTOR

Julio Cezar Neves é Analista de Suporte de Sistemas desde 1969 e trabalha com Unix desde 1980, quando fez parte da equipe que desenvolveu o SOX, sistema operacional, similar ao Unix, da Cobra Computadores. É professor do curso de Mestrado em Software Livre das Faculdades Estácio de Sá, no Rio de Janeiro.

Curso de Shell Script

Papo de Botequim - Parte II

Nossos personagens voltam à mesa do bar para discutir expressões regulares e colocar a "mão na massa" pela primeira vez, construindo um aplicativo simples para catalogar uma coleção de CDs. POR JÚLIO CÉSAR NEVES

Garçom! Traz um "chops" e dois "pastel". O meu amigo hoje não vai beber porque está finalmente sendo apresentado a um verdadeiro sistema operacional, e ainda tem muita coisa a aprender!

– E então, amigo, tá entendendo tudo que te expliquei até agora?
– Entendendo eu tô, mas não vi nada prático nisso…
– Calma rapaz, o que te falei até agora serve como base ao que há de vir daqui pra frente. Vamos usar essas ferramentas que vimos para montar programas estruturados. Você verá porque até na TV já teve programa chamado "O Shell é o Limite". Para começar vamos falar dos comandos da família *grep*
– Grep? Não conheço nenhum termo em inglês com este nome…
– É claro, grep é um acrônimo (sigla) para *Global Regular Expression Print*, que usa expressões regulares para pesquisar a ocorrência de cadeias de caracteres na entrada definida.

Por falar em expressões regulares (ou *regexp*), o Aurélio Marinho Jargas escreveu dois artigos [1 e 2] imperdíveis para a Revista do Linux sobre esse assunto e também publicou um livro [3] pela Editora Novatec. Acho bom você ler esses artigos, eles vão te ajudar no que está para vir.

Eu fico com grep, você com gripe

Esse negócio de gripe é brincadeira, só um pretexto para pedir umas caipirinhas. Eu te falei que o grep procura cadeias de caracteres dentro de uma entrada definida, mas o que vem a ser uma "entrada definida"? Bem, existem várias formas de definir a entrada do comando grep. Veja só. Para pesquisar em um arquivo:

```
$ grep franklin /etc/passwd
```

Pesquisando em vários arquivos:

```
$ grep grep *.sh
```

Pesquisando na saída de um comando:

```
$ who | grep carvalho
```

No 1º exemplo, procurei a palavra *franklin* em qualquer lugar do arquivo */etc/passwd*. Se quisesse procurar um nome de usuário, isto é, somente no início dos registros desse arquivo, poderia digitar *$ grep '^franklin' /etc/passwd*.

"E para que servem o circunflexo e os apóstrofos?", você vai me perguntar. Se tivesse lido os artigos que mencionei, saberia que o circunflexo serve para limitar a pesquisa ao início de cada linha e os apóstrofos servem para o Shell não interpretar esse circunflexo, deixando-o passar incólume para o comando *grep*.

No 2º exemplo mandei listar todas as linhas que usavam a palavra grep, em todos os arquivos terminados em *.sh*. Como uso essa extensão para definir meus arquivos com programas em Shell, malandramente, o que fiz foi listar as linhas dos programas que poderia usar como exemplo do comando *grep*.

Olha que legal! O grep aceita como entrada a saída de outro comando, redirecionado por um *pipe* (isso é muito comum em Shell e é um tremendo acelerador da execução de comandos). Dessa forma, no 3° exemplo, o comando *who* listou as pessoas "logadas" na mesma máquina que você (não se esqueça jamais: o Linux é multiusuário) e o *grep* foi usado para verificar se o Carvalho estava trabalhando ou "coçando". O *grep* é um comando muito conhecido, pois é usado com muita freqüência. O que muitas pessoas não sabem é que existem três comandos na família grep: *grep*, *egrep* e *fgrep*. A principais diferenças entre os 3 são:

- *grep* - Pode ou não usar expressões regulares simples, porém no caso de não usá-las, o *fgrep* é melhor, por ser mais rápido.
- *egrep* ("e" de *extended*, estendido) - É muito poderoso no uso de expressões regulares. Por ser o mais poderoso dos três, só deve ser usado quando for necessária a elaboração de uma expressão regular não aceita pelo *grep*.
- *fgrep* ("f" de *fast*, rápido) - Como o nome diz, é o ligeirinho da família, executando o serviço de forma muito veloz (por vezes é cerca de 30% mais rápido que o *grep* e 50% mais que o *egrep*), porém não permite o uso de expressões regulares na pesquisa.

–Agora que você já conhece as diferenças entre os membros da família, me diga: o que você acha dos três exemplos que eu dei antes das explicações?

– Achei que o *fgrep* resolveria o teu problema mais rapidamente que o *grep*.
– Perfeito! Tô vendo que você está atento, entendendo tudo que estou te explicando! Vamos ver mais exemplos

Quadro 1 - Listando subdiretórios

```
$ ls -l | grep '^d'
drwxr-xr-x    3       root    root    4096 Dec 18  2000 doc
drwxr-xr-x    11      root    root    4096 Jul 13 18:58 freeciv
drwxr-xr-x    3       root    root    4096 Oct 17  2000 gimp
drwxr-xr-x    3       root    root    4096 Aug  8  2000 gnome
drwxr-xr-x    2       root    root    4096 Aug  8  2000 idl
drwxrwxr-x    14      root    root    4096 Jul 13 18:58 locale
drwxrwxr-x    12      root    root    4096 Jan 14  2000 lyx
drwxrwxr-x    3       root    root    4096 Jan 17  2000 pixmaps
drwxr-xr-x    3       root    root    4096 Jul  2 20:30 scribus
drwxrwxr-x    3       root    root    4096 Jan 17  2000 sounds
drwxr-xr-x    3       root    root    4096 Dec 18  2000 xine
drwxr-xr-x    3       root    root    4096 Jun 19  2000 xplns
```

para clarear de vez as diferenças de uso entre os membros da família.

Eu sei que em um arquivo qualquer existe um texto falando sobre Linux, só não tenho certeza se está escrito com *L* maiúsculo ou minúsculo. Posso fazer uma busca de duas formas:

```
egrep (Linux | linux) arquivo.txt
```

ou então:

```
grep [Ll]inux arquivo.txt
```

No primeiro caso, a expressão regular complexa *(Linux | linux)* usa os parênteses para agrupar as opções e a barra vertical (|) é usada como um "ou" (*or*, em inglês) lógico, isto é, estou procurando *Linux* ou *linux*.

No segundo, a expressão regular *[Ll]inux* significa: começado por *L* ou *l* seguido de *inux*. Como esta é uma expressão simples, o *grep* consegue resolvê-la, por isso é melhor usar a segunda forma, já que o *egrep* tornaria a pesquisa mais lenta.

Outro exemplo. Para listar todos os subdiretórios do diretório corrente, basta usar o comando *$ ls -l | grep '^d'*. Veja o resultado no Quadro 1.

No exemplo, o circunflexo (^) serviu para limitar a pesquisa à primeira posição da saída do *ls* longo (parâmetro *-l*). Os apóstrofos foram usados para o Shell não "ver" o circunflexo. Vamos ver mais um. Veja na Tabela 1 as quatro primeiras posições possíveis da saída de um *ls -l* em um arquivo comum (não é diretório, nem link, nem ...).

Tabela 1

Posição	Valores possíveis
1ª	-
2ª	r ou -
3ª	w ou -
4ª	x, s(suid) ou -

Para descobrir todos os arquivos executáveis em um determinado diretório eu poderia fazer:

```
$ ls -la | egrep '^-..(x|s)'
```

novamente usamos o circunflexo para limitar a pesquisa ao início de cada linha, ou seja, listamos as linhas que começam por um traço (-), seguido de qualquer coisa (o ponto), novamente seguido de qualquer coisa, e por fim um *x* ou um *s*. Obteríamos o mesmo resultado se usássemos o comando:

```
$ ls -la | grep '^-..[xs]'
```

e além disso, agilizaríamos a pesquisa.

A "CDteca"

Vamos começar a desenvolver programas! Creio que a montagem de um banco de dados de músicas é bacana para efeito didático (e útil nestes tempos de downloads de arquivos MP3 e queimadores de CDs). Não se esqueça que, da mesma forma que vamos desenvolver um monte de programas para organizar os seus CDs de música, com pequenas adaptações você pode fazer o mesmo para organizar os CDs de software que vêm com a Linux Magazine e outros que você compra ou queima, e disponibilizar esse banco de software para todos os que trabalham com você (o Linux é multiusuário, e como tal deve ser explorado).

– Péra aí! De onde eu vou receber os dados dos CDs?
– Vou mostrar como o programa pode receber parâmetros de quem o estiver executando e, em breve, ensinarei a ler os dados da tela ou de um arquivo.

Passando parâmetros

Veja abaixo a estrutura do arquivo contendo a lista das músicas:

```
nomedoálbum^intérprete1~nome➥
damúsica1:...:intérpreten~nome➥
damúsican
```

Isto é, o nome do álbum será separado por um circunflexo do resto do registro, formado por diversos grupos compostos pelo intérprete de cada música do CD e a música interpretada. Estes grupos são separados entre si por dois pontos (:) e, internamente, o intérprete será separado por um til (~) do nome da música.

Quero escrever um programa chamado *musinc*, que incluirá registros no meu arquivo músicas. Passarei cada álbum como parâmetro para o programa:

```
$ musinc "álbum^interprete~➥
musica:interprete~musica:..."
```

Desta forma, musinc estará recebendo os dados de cada álbum como se fosse uma variável. A única diferença entre um parâmetro recebido e uma variável é que os primeiros recebem nomes numéricos (o que quis dizer é que seus nomes são formados somente por um algarismo, isto é, $1, $2, $3, ..., $9). Vamos, fazer mais alguns testes:

```
$ cat teste
#!/bin/bash
#Teste de passagem de parametros
echo "1o. parm -> $1"
echo "2o. parm -> $2"
echo "3o. parm -> $3"
```

Agora vamos rodar esse programinha:

```
$ teste passando parametros para ➥
testar
bash: teste: cannot execute
```

Ops! Esqueci-me de tornar o script executável. Vou fazer isso e testar novamente o programa:

```
$ chmod 755 teste
$ teste passando parametros para ➥
testar
1o. parm -> passando
2o. parm -> parametros
3o. parm -> para
```

Repare que a palavra testar, que seria o quarto parâmetro, não foi listada. Isso ocorreu porque o programa teste só lista os três primeiros parâmetros recebidos. Vamos executá-lo de outra forma:

```
$ teste "passando parametros" ➥
para testar
1o. parm -> passando parametros
2o. parm -> para
3o. parm -> testar
```

As aspas não deixaram o Shell ver o espaço em branco entre as duas primeiras palavras, e elas foram consideradas como um único parâmetro. E falando em passagem de parâmetros, uma dica: veja na Tabela 2 algumas variáveis especiais. Vamos alterar o programa *teste* para usar as novas variáveis:

```
$ cat teste
#!/bin/bash
# Programa para testar passagem ➥
de parametros (2a. Versao)
echo O programa $0 recebeu $# ➥
parametros
echo "1o. parm -> $1"
echo "2o. parm -> $2"
echo "3o. parm -> $3"
echo Para listar todos de uma ➥
\"tacada\" eu faco $*
```

Listagem 1: Incluindo CDs na "CDTeca"

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 1)
#
echo $1 >> musicas
```

Listagem 2

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 2)
#
echo $1 >> musicas
sort -o musicas musicas
```

Execute o programa:

```
$ teste passando parametros para testar
O programa teste recebeu 4 ➥
parametros
1o. parm -> passando
2o. parm -> parametros
3o. parm -> para
Para listar todos de uma ➥
"tacada" eu faco passando ➥
parametros para testar
```

Repare que antes das aspas usei uma barra invertida, para escondê-las da interpretação do Shell (se não usasse as contrabarras as aspas não apareceriam).

Como disse, os parâmetros recebem números de 1 a 9, mas isso não significa que não posso usar mais de nove parâmetros. Significa que só posso endereçar nove. Vamos testar isso:

```
$ cat teste
#!/bin/bash
# Programa para testar passagem ➥
de parametros (3a. Versao)
echo O programa $0 recebeu $# ➥
parametros
echo "11o. parm -> $11"
shift
echo "2o. parm -> $1"
shift 2
echo "4o. parm -> $1"
```

Execute o programa:

```
$ teste passando parametros para ➥
testar
O programa teste recebeu 4 ➥
parametros que são:
11o. parm -> passando1
2o. parm -> parametros
4o. parm -> testar
```

Duas coisas muito interessantes aconteceram neste script. Para mostrar que os nomes dos parâmetros variam de *$1* a *$9* digitei *echo $11* e o que aconteceu? O Shell interpretou como sendo *$1* seguido do algarismo 1 e listou *passando1*;

O comando *shift*, cuja sintaxe é *shift n*, podendo o *n* assumir qualquer valor numérico, despreza os *n* primeiros parâmetros, tornando o parâmetro de ordem *n + 1*.

Bem, agora que você já sabe sobre passagem de parâmetros, vamos voltar à nossa "cdteca" para fazer o script de inclusão de CDs no meu banco chamado musicas. O programa é muito simples (como tudo em Shell). Veja a Listagem 1.

O script é simples e funcional; limito-me a anexar ao fim do arquivo *musicas* o parâmetro recebido. Vamos cadastrar 3 álbuns para ver se funciona (para não ficar "enchendo lingüiça," suponho que em cada CD só existem duas músicas):

```
$ musinc "album3^Artista5➥
~Musica5:Artista6~Musica5"
$ musinc "album1^Artista1➥
~Musica1:Artista2~Musica2"
$ musinc "album 2^Artista3➥
~Musica3:Artista4~Musica4"
```

Listando o conteúdo do arquivo *musicas*:

```
$ cat musicas
album3^Artista5~Musica5:Artista6➥
~Musica6
album1^Artista1~Musica1:Artista2➥
~Musica2
album2^Artista3~Musica3:Artista4➥
~Musica4
```

Podia ter ficado melhor. Os álbuns estão fora de ordem, dificultando a pesquisa. Vamos alterar nosso script e depois testá-lo novamente. Veja a listagem 2. Simplesmente inseri uma linha que classifica o arquivo *musicas*, redirecionando a saída para ele mesmo (para isso serve a opção *-o*), após cada álbum ser anexado.

```
$ cat musicas
album1^Artista1~Musica1:Artista2➥
~Musica2
albu2^Artista3~Musica3:Artista4➥
~Musica4
album3^Artista5~Musica5:Artista6➥
~Musica6
```

Oba! Agora o programa está legal e quase funcional. Ficará muito melhor em uma nova versão, que desenvolveremos após aprender a adquirir os dados da tela e formatar a entrada.

Tabela 2: Variáveis especiais

Variável	Significado
$0	Contém o nome do programa
$#	Contém a quantidade de parâmetros passados
$*	Contém o conjunto de todos os parâmetros (muito parecido com $@)

Ficar listando arquivos com o comando *cat* não está com nada, vamos fazer um programa chamado *muslist* para listar um álbum, cujo nome será passado como parâmetro. Veja o código na Listagem 3:

Vamos executá-lo, procurando pelo album 2. Como já vimos antes, para passar a string *album 2* é necessário protegê-la da interpretação do Shell, para que ele não a interprete como dois parâmetros. Exemplo:

```
$ muslist "album 2"
grep: can't open 2
musicas: album1^Artista1~Musica1
:Artista2~Musica2
musicas: album2^Artista3~Musica3
:Artista4~Musica4
musicas:album3^Artista5~Musica5
:Artista6~Musica6
```

Que lambança! Onde está o erro? Eu tive o cuidado de colocar o parâmetro passado entre aspas para o Shell não o dividir em dois! É, mas repare como o grep está sendo executado:

```
grep $1 musicas
```

Mesmo colocando *álbum 2* entre aspas, para que fosse encarado como um único parâmetro, quando o *$1* foi passado pelo Shell para o comando *grep*, transformou-se em dois argumentos. Dessa forma, o conteúdo da linha que o grep executou foi o seguinte:

```
grep album 2 musicas
```

Como a sintaxe do grep é:

Listagem 3 - muslist

```
$ cat muslist
#!/bin/bash
# Consulta CDs (versao 1)
#
grep $1 musicas
```

Listagem 4 muslist melhorado

```
$ cat muslist
#!/bin/bash
# Consulta CDs (versao 2)
#
grep -i "$1" musicas
```

```
grep <cadeia de caracteres>
[arq1, arq2, ..., arqn]
```

O *grep* entendeu que deveria procurar a cadeia de caracteres album nos arquivos *2* e *musicas*. Como o arquivo 2 não existe, grep gerou o erro e, por encontrar a palavra album em todos os registros de musicas, listou a todos.

É melhor ignorarmos maiúsculas e minúsculas na pesquisa. Resolveremos os dois problemas com a Listagem 4.

Nesse caso, usamos a opção *-i* do grep que, como já vimos, serve para ignorar maiúsculas e minúsculas, e colocamos o *$1* entre aspas, para que o grep continuasse a ver a cadeia de caracteres resultante da expansão da linha pelo Shell como um único argumento de pesquisa.

```
$ muslist "album 2"
album2^Artista3~Musica3:Artista4
~Musica4
```

Agora repare que o grep localiza a cadeia pesquisada em qualquer lugar do registro; então, da forma que estamos fazendo, podemos pesquisar por álbum, por música, por intérprete e mais. Quando conhecermos os comandos condicionais, montaremos uma nova versão de muslist que permitirá especificar por qual campo pesquisar.

Ah! Em um dia de verão você foi à praia, esqueceu os CDs no carro, aquele "solzinho" de 40 graus empenou seu disco favorito e agora você precisa de uma ferramenta para removê-lo do banco de dados? Não tem problema, vamos desenvolver um script chamado *musexc*, para excluir estes CDs.

Antes de desenvolver o "bacalho", quero te apresentar a uma opção bastante útil da família de comandos grep. É a opção *-v*, que quando usada lista todos os registros da entrada, exceto o(s) localizado(s) pelo comando. Exemplos:

```
$ grep -v "album 2" musicas
album1^Artista1~Musica1:Artista2
~Musica2
album3^Artista5~Musica5:Artista6
~Musica6
```

Conforme expliquei antes, o grep do exemplo listou todos os registros de musicas exceto o referente a album 2, porque atendia ao argumento do comando. Estamos então prontos para desenvolver o script para remover CDs empenados da sua "CDteca". Veja o código da Listagem 5.

Listagem 5 - musexc

```
$ cat musexc
#!/bin/bash
# Exclui CDs (versao 1)
#
grep -v "$1" musicas > /tmp/mus$$
mv -f /tmp/mus$$ musicas
```

Na primeira linha mandei para */tmp/mus$$* o arquivo musicas, sem os registros que atendessem a consulta feita pelo comando grep. Em seguida, movi */tmp/mus$$* por cima do antigo *musicas*. Usei o arquivo */tmp/mus$$* como arquivo de trabalho porque, como já havia citado no artigo anterior, o *$$* contém o PID (identificação do processo) e, dessa forma, cada um que editar o arquivo musicas o fará em um arquivo de trabalho diferente, evitando colisões.

Os programas que fizemos até aqui ainda são muito simples, devido à falta de ferramentas que ainda temos. Mas é bom praticar os exemplos dados porque, eu prometo, chegaremos a desenvolver um sistema bacana para controle dos seus CDs. Na próxima vez que nos encontrarmos, vou te ensinar como funcionam os comandos condicionais e aprimoraremos mais um pouco esses scripts. Por hoje chega! Já falei demais e estou de goela seca! Garçom! Mais um sem colarinho!

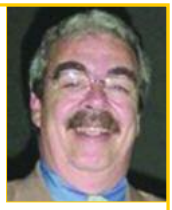
INFORMAÇÕES

[1] *http://www.revistadolinux.com.br/ed/003/ferramentas.php3*

[2] *http://www.revistadolinux.com.br/ed/007/ereg.php3*

[3] *http://www.aurelio.net/er/livro/*

SOBRE O AUTOR



Julio Cezar Neves é Analista de Suporte de Sistemas desde 1969 e trabalha com Unix desde 1980, quando fez parte da equipe que desenvolveu o SOX, um sistema operacional similar ao Unix, produzido pela Cobra Computadores.

Curso de Shell Script

Papo de botequim III

Um chopinho, um aperitivo e o papo continua. Desta vez vamos aprender alguns comandos de manipulação de cadeias de caracteres, que serão muito úteis na hora de incrementar nossa "CDteca". POR JULIO CEZAR NEVES

Garçon! traga dois chopes por favor que hoje eu vou ter que falar muito. Primeiro quero mostrar uns programinhas simples de usar e muito úteis, como o *cut*, que é usado para cortar um determinado pedaço de um arquivo. A sintaxe e alguns exemplos de uso podem ser vistos no Quadro 1:

Como dá para ver, existem quatro sintaxes distintas: na primeira (*-c 1-5*) especifiquei uma faixa, na segunda (*-c -6*) especifiquei todo o texto até uma posição, na terceira (*-c 4-*) tudo de uma determinada posição em diante e na quarta (*-c 1,3,5,7,9*), só as posições determinadas. A última possibilidade (*-c -3,5,8-*) foi só para mostrar que podemos misturar tudo.

Mas não pense que acabou por aí! Como você deve ter percebido, esta forma de cut é muito útil para lidar com arquivos com campos de tamanho fixo, mas atualmente o que mais existe são arquivos com campos de tamanho variável, onde cada campo termina com um delimitador. Vamos dar uma olhada no arquivo *musicas* que começamos a preparar na última vez que viemos aqui no botequim. Veja o Quadro 2.

Então, recapitulando, o layout do arquivo é o seguinte: *nome do álbum^intérprete1~nome da música1:...: intérpreten~nome da músican*, isto é, o nome do álbum será separado por um circunflexo (^) do resto do registro, que é formado por diversos grupos compostos pelo intérprete de cada música do CD e a respectiva música interpretada. Estes grupos são separados entre si por dois-pontos (:) e o intérprete será separado do nome da música por um til (~).

Então, para pegar os dados referentes a todas as segundas músicas do arquivo musicas, devemos digitar:

```
$ cut -f2 -d: musicas
Artista2~Musica2
Artista4~Musica4
Artista6~Musica5
Artista8~Musica8@10_L:
```

Ou seja, cortamos o segundo campo z(*-f* de *field*, campo em inglês) delimitado (*-d*) por dois-pontos (*:*). Mas, se quisermos somente os intérpretes, devemos digitar:

```
$ cut -f2 -d: musicas | cut ➥
-f1 -d~
Artista2
Artista4
Artista6
Artista8
```

Para entender melhor isso, vamos analisar a primeira linha de musicas:

```
$ head -1 musicas
album 1^Artista1~Musica1: ➥
Artista2~Musica2
```

Então observe o que foi feito:

```
album 1^Artista1~Musica1: ➥
Artista2~Musica2
```

Desta forma, no primeiro cut o primeiro campo do delimitador (*-d*) dois-pontos (*:*) é *album 1^Artista1~Musica1* e o segundo, que é o que nos interessa, é *Artista2~Musica2*. Vamos então ver o que aconteceu no segundo cut:

```
Artista2~Musica2
```

Agora, primeiro campo do delimitador (*-d*) til (~), que é o que nos interessa, é *Artista2* e o segundo é *Musica2*. Se o raciocínio que fizemos para a pri-

Quadro 1 – O comando cut

A sintaxe do cut é: *cut -c PosIni-PosFim [arquivo]*, onde *PosIni* é a posição inicial, e *PosFim* a posição final. Veja os exemplos:

```
$ cat numeros
1234567890
0987654321
1234554321
9876556789

$ cut -c1-5 numeros
12345
09876
12345
98765

$ cut -c-6 numeros
123456
098765
123455
987655

$ cut -c4- numeros
4567890
7654321
4554321
6556789

$ cut -c1,3,5,7,9 numeros
13579
08642
13542
97568

$ cut -c -3,5,8- numeros
1235890
0986321
1235321
9875789
```

Quadro 2 – O arquivo *musicas*

```
$ cat musicas
album 1^Artista1~Musica1:➥
Artista2~Musica2
album 2^Artista3~Musica3:➥
Artista4~Musica4
album 3^Artista5~Musica5:➥
Artista6~Musica5
album 4^Artista7~Musica7:➥
Artista8~Musica8
```

meira linha for aplicado ao restante do arquivo, chegaremos à resposta anteriormente dada. Outro comando muito interessante é o *tr* que serve para substituir, comprimir ou remover caracteres. Sua sintaxe segue o seguinte padrão:

```
tr [opções] cadeia1 [cadeia2]
```

O comando copia o texto da entrada padrão (*stdin*), troca as ocorrência dos caracteres de *cadeia1* pelo seu correspondente na *cadeia2* ou troca múltiplas ocorrências dos caracteres de *cadeia1* por somente um caracter, ou ainda caracteres da *cadeia1*. As principais opções do comando são mostradas na Tabela 1.

Primeiro veja um exemplo bem bobo:

```
$ echo bobo | tr o a
baba
```

Isto é, troquei todas as ocorrências da letra *o* pela letra *a*. Suponha que em determinado ponto do meu script eu peça ao operador para digitar *s* ou *n* (sim ou não), e guardo sua resposta na variável *$Resp*. Ora, o conteúdo de *$Resp* pode conter letras maiúsculas ou minúsculas, e desta forma eu teria que fazer diversos testes para saber se a resposta dada foi *S, s, N* ou *n*. Então o melhor é fazer:

```
$ Resp=$(echo $Resp | tr SN sn)
```

e após este comando eu teria certeza de que o conteúdo de *$Resp* seria um *s* ou um *n*. Se o meu arquivo *ArqEnt* está todo em letras maiúsculas e desejo passá-las para minúsculas eu faço:

```
$ tr A-Z a-z < ArqEnt > / tmp/$$
$ mv -f /tmp/$$ ArqEnt
```

Note que neste caso usei a notação A-Z para não escrever ABCD...YZ. Outro tipo de notação que pode ser usada são as *escape sequences* (como eu traduziria? Seqüências de escape? Meio sem sentido, né? Mas vá lá...) que também são reconhecidas por outros comandos e também na linguagem C, e cujo significado você verá na Tabela 2:

Deixa eu te contar um "causo": um aluno que estava danado comigo resolveu complicar minha vida e como resposta a um exercício prático, valendo nota, que passei ele me entregou um script com todos os comandos separados por ponto-e-vírgula (lembre-se que o ponto-e-vírgula serve para separar diversos comandos em uma mesma linha). Vou dar um exemplo simplificado, e idiota, de um script assim:

```
$ cat confuso
echo leia Programação Shell ➥
Linux do Julio Cezar Neves ➥
> livro;cat livro;pwd;ls;rm ➥
-f livro2>/dev/null;cd ~
```

Eu executei o programa e ele funcionou:

```
$ confuso
leia Programação Shell Linux ➥
do Julio Cezar Neves
/home/jneves/LM
confuso  livro  musexc  musicas
musinc  muslist  numeros
```

Mas nota de prova é coisa séria (e nota de dólar é mais ainda) então, para entender o que o aluno havia feito, o chamei e em sua frente digitei:

```
$ tr ";" "\n" < confuso
echo leia Programação Shell ➥
Linux do Julio Cezar Neves
pwd
cd ~
ls -l
rm -f lixo 2>/dev/null
```

O cara ficou muito desapontado, porque em dois ou três segundos eu desfiz a gozação que ele perdeu horas para fazer. Mas preste atenção! Se eu estivesse em uma máquina Unix, eu teria digitado:

```
$ tr ";" "\012" < confuso
```

Agora veja a diferença entre o resultado de um comando date executado hoje e outro executado há duas semanas:

```
Sun Sep 19 14:59:54  2004
Sun Sep  5 10:12:33  2004
```

Notou o espaço extra após o "Sep" na segunda linha? Para pegar a hora eu deveria digitar:

```
$ date | cut -f 4 -d ' '
14:59:54
```

Mas há duas semanas ocorreria o seguinte:

```
$ date | cut -f 4 -d ' '
5
```

Isto porque existem 2 caracteres em branco antes do 5 (dia). Então o ideal seria transformar os espaços em branco consecutivos em somente um espaço para poder tratar os dois resultados do comando *date* da mesma forma, e isso se faz assim:

```
$ date | tr -s " "
Sun Sep 5 10:12:33 2004
```

E agora eu posso cortar:

```
$ date | tr -s " " | cut -f 4 ➥
-d " "
10:12:33
```

Olha só como o Shell está quebrando o galho. Veja o conteúdo de um arquivo baixado de uma máquina Windows:

```
$  cat -ve ArqDoDOS.txt
Este arquivo^M$
foi gerado pelo^M$
DOS/Win e foi^M$
baixado por um^M$
ftp mal feito.^M$
```

Dica: a opção *-v* do cat mostra os caracteres de controle invisíveis, com a notação ^*L*, onde ^ é a tecla *Control* e *L* é a respectiva letra. A opção *-e* mostra o final da linha como um cifrão (*$*).

Isto ocorre porque no DOS o fim dos registros é indicado por um *Carriage Return* (\r – Retorno de Carro, CR) e um *Line Feed* (\f – Avanço de Linha, ou LF). No Linux porém o final do registro é indicado somente pelo *Line Feed*. Vamos limpar este arquivo:

```
$ tr -d '\r' < ArqDoDOS.txt > /tmp/$$
$ mv -f /tmp/$$ ArqDoDOS.txt
```

Agora vamos ver o que aconteceu:

```
$ cat -ve ArqDoDOS.txt
Este arquivo$
foi gerado pelo$
DOS/Rwin e foi$
baixado por um$
ftp mal feito.$
```

Bem a opção *-d* do *tr* remove do arquivo todas as ocorrências do caractere especificado. Desta forma eu removi os caracteres indesejados, salvei o texto em um arquivo temporário e posteriormente renomeei-o para o nome original. Uma observação: em um sistema Unix eu deveria digitar:

```
$ tr -d '\015' < ArqDoDOS.➥
txt > /tmp/$$
```

Uma dica: o problema com os terminadores de linha (CR/LF) só aconteceu porque a transferência do arquivo foi feita no modo binário (ou image), Se antes da transmissão do arquivo tivesse sido estipulada a opção *ascii* do ftp, isto não teria ocorrido.

– Olha, depois desta dica tô começando a gostar deste tal de shell, mas ainda tem muita coisa que não consigo fazer.

– Pois é, ainda não te falei quase nada sobre programação em shell, ainda tem muita coisa para aprender, mas com o que aprendeu, já dá para resolver muitos problemas, desde que você adquira o "modo shell de pensar". Você seria capaz de fazer um script que diga quais pessoas estão "logadas" há mais de um dia no seu servidor?

– Claro que não! Para isso seria necessário eu conhecer os comandos condicionais que você ainda não me explicou como funcionam. - Deixa eu tentar mudar um pouco a sua lógica e trazê-la para o "modo shell de pensar", mas antes é melhor tomarmos um chope.

Agora que já molhei a palavra, vamos resolver o problema que te propus. Veja como funciona o comando *who*:

```
$ who
jneves   pts/➥
1        Sep 18 13:40
rtorres  pts/➥
0        Sep 20 07:01
rlegaria pts/➥
1        Sep 20 08:19
lcarlos  pts/➥
3        Sep 20 10:01
```

Tabela 1 – O comando tr

Opção	Significado
-s	Comprime n ocorrências de cadeia1 em apenas uma
-d	Remove os caracteres de cadeia1

E veja também o date:

```
$ date
Mon Sep 20 10:47:19 BRT 2004
```

Repare que o mês e o dia estão no mesmo formato em ambos os comandos. Ora, se em algum registro do *who* eu não encontrar a data de hoje, é sinal que o usuário está "logado" há mais de um dia, já que ele não pode ter se "logado" amanhã... Então vamos guardar o pedaço que importa da data de hoje para depois procurá-la na saída do comando *who*:

```
$ Data=$(date | cut -f 2-3 ➥
-d' ')
```

Eu usei a construção *$(...)*, para priorizar a execução dos comandos antes de atribuir a sua saída à variável *Data*. Vamos ver se funcionou:

```
$ echo $Data
Sep 20
```

Beleza! Agora, o que temos que fazer é procurar no comando who os registros que não possuem esta data.

– Ah! Eu acho que estou entendendo! Você falou em procurar e me ocorreu o comando grep, estou certo?

– Certíssimo! Só que eu tenho que usar o grep com aquela opção que ele só lista os registros nos quais ele não encontrou a cadeia. Você se lembra que opção é essa?

– Claro, a *-v*...

– Isso! Tá ficando bom! Vamos ver:

```
$  who | grep -v "$Data"
jneves   pts/➥
1        Sep 18 13:40
```

Se eu quisesse um pouco mais de perfumaria eu faria assim:

```
$  who | grep -v "$Data" |➥
 cut -f1 -d ' '
jneves
```

Viu? Não foi necessário usar comando condicional, até porque o nosso comando condicional, o famoso if, não testa condição, mas sim instruções. Mas antes veja isso:

```
$ ls musicas
musicas
$ echo $?
0
$ ls ArqInexistente
ls: ArqInexistente: No such ➥
file or directory
$ echo $?
1
$ who | grep jneves
jneves   pts/1        Sep 18 ➥
13:40 (10.2.4.144)
$ echo $?
0
$ who | grep juliana
$ echo $?
1
```

O que é que esse *$?* faz aí? Algo começado por cifrão ($) parece ser uma variável, certo? Sim é uma variável que contém o código de retorno da última instrução executada. Posso te garantir que se esta instrução foi bem sucedida, *$?* terá o valor zero, caso contrário seu valor será diferente de zero. O que nosso comando condicional (if) faz é testar esta variável. Então vamos ver a sua sintaxe:

```
if cmd
then
    cmd1
    cmd2
    cmdn
else
    cmd3
    cmd4
    cmdm
fi
```

Ou seja, caso comando *cmd* tenha sido executado com sucesso, os comandos do bloco *do then* (*cmd1, cmd2 e cmdn*) serão executados, caso contrário, os comandos do bloco opcional *do else* (*cmd3, cmd4 e cmdm*) serão executados. O bloco do *if* é terminando com um *fi*.

Vamos ver na prática como isso funciona, usando um script que inclui usuários no arquivo */etc/passwd*:

```
$ cat incusu
#!/bin/bash
# Versão 1
if  grep ^$1 /etc/passwd
then
    echo Usuario \'$1\'➥
```

Tabela 2

Seqüência	Significado	Octal
\t	Tabulação	\011
\n	Nova linha <ENTER>	\012
\v	Tabulação Vertical	\013
\f	Nova Página	\014
\r	Início da linha <^M>	\015
\\	Uma barra invertida	\0134

```
    já existe
else
    if  useradd $1
    then
        echo Usuário \'$1\' ➥
incluído em /etc/passwd
    else
        echo "Problemas no ➥
cadastramento. Você é root?"
    fi
fi
```

Repare que o *if* está testando direto o comando *grep* e esta é a sua finalidade. Caso o *if* seja bem sucedido, ou seja, o usuário (cujo nome está em *$1*) foi encontrado em */etc/passwd*, os comandos do bloco *do then* serão executados (neste exemplo, apenas o echo). Caso contrário, as instruções do bloco *do else* serão executadas, quando um novo *if* testa se o comando *useradd* foi executado a contento, criando o registro do usuário em */etc/passwd*, ou exibindo uma mensagem de erro, caso contrário.

Executar o programa e passe como parâmetro um usuário já cadastrado:

```
$ incusu jneves
jneves:x:54002:1001:Julio Neves:➥
/home/jneves:/bin/➥
bash
Usuario 'jneves' ja existe
```

No exemplo dado, surgiu uma linha indesejada, ela é a saída do comando grep. Para evitar que isso aconteça, devemos desviar a saída para */dev/null*. O programa fica assim:

```
$ cat incusu
#!/bin/bash
# Versão 2
if  grep ^$1 /etc/passwd > ➥
/dev/null
then
    echo Usuario \'$1\' já ➥
existe
else
    if  useradd $1
    then
        echo Usuário \'$1\' ➥
incluído em /etc/passwd
    else
        echo "Problemas no ➥
cadastramento. Você é root?"
    fi
fi
```

Vamos testá-lo como um usuário normal :

```
$ incusu ZeNinguem
./incusu[6]: useradd: not found
 Problemas no cadastramento. ➥
 Você é root?
```

Aquela mensagem de erro não deveria aparecer! Para evitar isso, devemos redirecionar a saída de erro (*stderr*) do comando *useradd* para */dev/null*. A versão final fica assim:

```
$ cat incusu
#!/bin/bash
# Versão 3
if  grep ^$1 /etc/passwd > ➥
/dev/null
then
    echo Usuario \'$1\' já ➥
existe
else
    if  useradd $1 2> /dev/null
    then
        echo Usuário \'$1\' ➥
incluído em /etc/passwd
    else
        echo "Problemas no ➥
cadastramento. Você é root?"
    fi
fi
```

Depois disso, vejamos o comportamento do programa, se executado pelo root:

```
$ incusu botelho
Usuário 'botelho' incluido em ➥
/etc/passwd
```

E novamente:

```
$ incusu botelho
Usuário 'botelho' já existe
```

Lembra que eu falei que ao longo dos nossos papos e chopes os nossos programas iriam se aprimorando? Então vejamos agora como podemos melhorar o nosso programa para incluir músicas na "CDTeca":

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 3)
#
if  grep "^$1$" musicas > ➥
/dev/null
then
    echo Este álbum já está ➥
cadastrado
else
    echo $1 >> musicas
    sort musicas -o musicas
fi
```

Como você viu, é uma pequena evolução em relação à versão anterior. Antes de incluir um registro (que na versão anterior poderia ser duplicado), testamos se o registro começa (^) e termina ($) de forma idêntica ao parâmetro *álbum* passado ($1). O circunflexo (^) no início da cadeia e cifrão ($) no fim, servem para testar se o parâmetro (o álbum e seus dados) é exatamente igual a algum registro já existente. Vamos executar nosso programa novamente, mas desta vez passamos como parâmetro um álbum já cadastrado, pra ver o que acontece:

```
$ musinc "album 4^Artista7~➥
Musica7:Artista8~Musica8"
Este álbum já está cadastrado
```

E agora um não cadastrado:

```
$ musinc "album 5^Artista9~➥
Musica9:Artista10~Musica10"
$ cat musicas
album 1^Artista1~Musica1:➥
Artista2~Musica2
album 2^Artista3~Musica3:➥
Artista4~Musica4
album 3^Artista5~Musica5:➥
Artista6~Musica5
album 4^Artista7~Musica7:➥
Artista8~Musica8
album 5^Artista9~Musica9:➥
Artista10~Musica10
```

Como você viu, o programa melhorou um pouquinho, mas ainda não está pronto. À medida que eu te ensinar a programar em shell, nossa CDteca vai ficar cada vez melhor.

– Entendi tudo que você me explicou, mas ainda não sei como fazer um if para testar condições, ou seja o uso normal do comando.

– Para isso existe o comando test, que testa condições. O comando if testa o comando test. Como já falei muito, preciso de uns chopes para molhar a palavra. Vamos parar por aqui e na próxima vez te explico direitinho o uso do test e de diversas outras sintaxes do if.

– Falou! Acho bom mesmo porque eu também já tô ficando zonzo e assim tenho tempo para praticar esse monte de coisas que você me falou hoje.

Para fixar o que você aprendeu, tente fazer um scriptizinho para informar se um determinado usuário, cujo nome será passado como parâmetro, está "logado" no sistema ou não.

– Aê Chico! traz mais dois chopes pra mim por favor...

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.

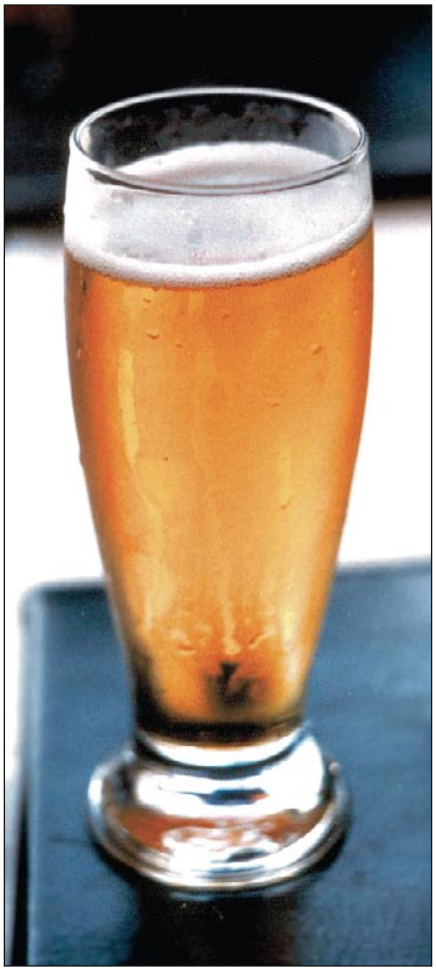
Para mostrar como o código fica melhor, vamos repetir o exemplo anterior, só que desta vez usaremos o *case* em vez do tradicional bloco de código com *if ... elif ... else ... fi*.

```
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


Dave Hamilton - www.sxc.hu

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~Musica1:➥
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 *Artistan~Musicam*. 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~Musica1
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 musica1 ➥
[musica2] ... [musican]
    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


Dave Hamilton - www.sxc.hu

Curso de Shell Script

Papo de Botequim VI

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 aí, já tá sabendo tudo do comando *for*? Eu te deixei um exercício para treinar, se não me engano era para contar a quantidade de palavras de um arquivo... Você fez?

– Claro! Tô empolgadão com essa linguagem! Eu fiz da forma que você pediu, olha só...
– Êpa! Peraí que eu tô sequinho pra tomar um chope. Aê Chico, traz dois por favor. Um sem colarinho!
– Como eu ia dizendo, olha como eu fiz. É muito fácil...

```
$ cat contpal.sh
#!/bin/bash
# Script meramente pedagógico
# cuja função é contar a
# quantidade de palavras de
# um arquivo. Supõe-se que as
# palavras estão separadas
# entre si por espaços, <TAB>
# ou <ENTER>.
if  [ $# -ne 1 ]
then
    echo uso: $0 /caminho/do/➥
arquivo
    exit 2
fi
Cont=0
for Palavra in $(cat $1)
do
    Cont=$((Cont+1))
done
echo O arquivo $1 tem $Cont ➥
palavras.
```

Ou seja, o programa começa, como sempre, verificando se a passagem de parâmetros foi correta; em seguida o comando *for* se incumbe de pegar cada uma das palavras (lembre-se que o *$IFS* padrão é branco, *TAB* e *ENTER*, que é exatamente o que desejamos para separar as palavras), incrementando a variável *$Cont*. Vamos relembrar como é o arquivo *ArqDoDOS.txt*.

```
$ cat ArqDoDOS.txt
Este arquivo
foi gerado pelo
DOS/Rwin e foi
baixado por um
ftp mal feito.
```

Agora vamos testar o programa passando esse arquivo como parâmetro:

```
$ contpal.sh ArqDoDOS.txt
O arquivo ArqDoDOS.txt tem 14
palavras.
```

Funcionou legal! Se você se lembra, em nossa última aula mostramos o loop *for* a seguir:

```
for ((; i<=9;))
do
    let i++
    echo -n "$i "
done
```

Uma vez que chegamos neste ponto, creio ser interessante citar que o Shell trabalha com o conceito de "Expansão Aritmética" (*Arithmetic Expansion*), que é acionada por uma construção da forma *$((expressão))* ou *let expressão*.

No último loop *for* usei a expansão aritmética das duas formas, mas não podemos seguir adiante sem saber que a expressão pode ser uma das listadas na tabela 1.

Mas você pensa que o papo de loop (ou laço) se encerra no comando *for*? Ledo engano, amigo, vamos a partir de agora ver mais dois comandos.

O comando *while*

Todos os programadores conhecem este comando, porque é comum a todas as linguagens. Nelas, o que normalmente ocorre é que um bloco de comandos é executado, enquanto (enquanto, em inglês, é "while") uma determinada condição for verdadeira.

Tabela 1: Expressões no Shell

Expressão	Resultado
id++ id--	pós-incremento e pós-decremento de variáveis
++id --id	pré-incremento e pré-decremento de variáveis
**	exponenciação
* / %	multiplicação, divisão, resto da divisão (módulo)
+ -	adição, subtração
<= >= < >	comparação
== !=	igualdade, desigualdade
&&	E lógico
||	OU lógico

Pois bem, isso é o que acontece nas linguagens caretas! Em programação Shell, o bloco de comandos é executado enquanto um comando for verdadeiro. E é claro, se quiser testar uma condição, use o comando *while* junto com o comando *test*, exatamente como você aprendeu a fazer no *if*, lembra? Então a sintaxe do comando fica assim:

```
while comando
do
    cmd1
    cmd2
    ...
    cmdn
done
```

e dessa forma, o bloco formado pelas instruções *cmd1, cmd2,...* e *cmdn* é executado enquanto a execução da instrução *comando* for bem sucedida.

Suponha a seguinte cena: tinha uma tremenda gata me esperando e eu estava preso no trabalho sem poder sair porque o meu chefe, que é um pé no saco (aliás chefe-chato é uma redundância, né?), ainda estava na sala dele, que fica bem na minha passagem para a rua. Ele começou a ficar cabreiro depois da quinta vez que passei pela sua porta e olhei para ver se já havia ido embora. Então voltei para a minha mesa e fiz, no servidor, um script assim:

```
$ cat logaute.sh
#!/bin/bash
# Espero que a Xuxa não tenha
# copyright de xefe e xato :)
while who | grep xefe
do
    sleep 30
done
echo O xato se mandou, não ➥
hesite, dê exit e vá à luta
```

Neste scriptzinho, o comando *while* testa o pipeline composto pelos comandos *who* e *grep*, que será verdadeiro enquanto o *grep* localizar a palavra *xefe* na saída do comando *who*. Desta forma, o script dormirá por 30 segundos enquanto o chefe estiver logado (Argh!). Assim que ele se desconectar do servidor, o fluxo do script sairá do loop e te mostrará a tão ansiada mensagem de liberdade. Mas quando executei o script, adivinha o que aconteceu?

```
$ logaute.sh
xefe  pts/0   Jan  4 08:46 ➥
(10.2.4.144)
xefe  pts/0   Jan  4 08:46 ➥
(10.2.4.144)
...
xefe  pts/0   Jan  4 08:46 ➥
(10.2.4.144)
```

Isto é, a cada 30 segundos a saída do comando grep seria enviada para a tela, o que não é legal, já que poluiria a tela do meu micro e a mensagem tão esperada poderia passar despercebida. Para evitar isso, já sabemos que a saída do pipeline tem que ser redirecionada para o dispositivo */dev/null*.

```
$ cat logaute.sh
#!/bin/bash
# Espero que a Xuxa não tenha
# copyright de xefe e xato :)
while who | grep xefe > /dev/null
do
    sleep 30
done
echo O xato se mandou, não ➥
hesite, dê exit e vá a luta
```

Agora quero montar um script que receba o nome (e eventuais parâmetros) de um programa que será executado em background e que me informe do seu término. Mas, para você entender este exemplo, primeiro tenho de mostrar uma nova variável do sistema. Veja estes comandos executados diretamente no prompt:

```
$ sleep 10&
[1] 16317
$ echo $!
16317
[1]+  Done                    sleep 10
$ echo $!
16317
```

Isto é, criei um processo em background que dorme por 10 segundos, somente para mostrar que a variável *$!* guarda o PID (*Process ID*) do último processo em background. Mas observe a listagem e repare, após a linha do *Done*, que a variável reteve o valor mesmo após o término desse processo.

Bem, sabendo isso, já fica mais fácil monitorar qualquer processo em background. Veja só como:

```
$cat monbg.sh
#!/bin/bash
# Executa e  monitora um
# processo em background
$1 &     # Coloca em backgroud
while ps | grep -q $!
do
    sleep 5
done
echo Fim do Processo $1
```

Esse script é bastante similar ao anterior, mas tem uns macetes a mais, veja só: ele tem que ser executado em background para não prender o prompt mas o *$!* será o do programa passado como parâmetro, já que ele foi colocado em background após o *monbg.sh* propriamente dito. Repare também na opção *-q* (quiet) do grep, que serve para fazê-lo "trabalhar em silêncio". O mesmo resultado poderia ser obtido com a linha: *while ps | grep $! > /dev/null*, como nos exemplos que vimos até agora.

Vamos melhorar o nosso velho *musinc*, nosso programa para incluir registros no arquivo *musicas*, mas antes preciso te ensinar a pegar um dado da tela, e já vou avisando: só vou dar uma pequena dica do comando read (que é quem pega o dado da tela), que seja o suficiente para resolver este nosso problema. Em uma outra rodada de chope vou te ensinar tudo sobre o assunto, inclusive como formatar tela, mas hoje estamos falando sobre loops. A sintaxe do comando read que nos interessa por hoje é a seguinte:

```
$ read -p "prompt de leitura" var
```

Onde "prompt de leitura" é o texto que você quer que apareça escrito na tela. Quando o operador teclar tal dado, ele será armazenado na variável *var*. Por exemplo:

```
$ read -p "Título do Álbum: " Tit
```

Bem, uma vez entendido isso, vamos à especificação do nosso problema: faremos um programa que inicialmente lerá o nome do álbum e em seguida fará um loop de leitura, pegando o nome da música e o artista. Esse loop termina quando for informada uma música com nome vazio, isto é, quando o operador

Dica

Leitura de arquivo significa ler um a um todos os registros, o que é sempre uma operação lenta. Fique atento para não usar o while quando for desnecessário. O Shell tem ferramentas como o sed e a família grep, que vasculham arquivos de forma otimizada sem que seja necessário o uso do while para fazê-lo registro a registro.

der um simples <ENTER>. Para facilitar a vida do operador, vamos oferecer como default o mesmo nome do artista da música anterior (já que é normal que o álbum seja todo do mesmo artista) até que ele deseje alterá-lo. Veja na listagem 1 como ficou o programa.

Nosso exemplo começa com a leitura do título do álbum. Caso ele não seja informado, terminamos a execução do programa. Em seguida um *grep* procura, no início (^) de cada registro de músicas, o título informado seguido do separador (^) (que está precedido de uma contrabarra [\] para protegê-lo da interpretação do Shell).

Para ler os nomes dos artistas e as músicas do álbum, foi montado um loop *while* simples, cujo único destaque é o fato de ele armazenar o nome do intérprete da música anterior na variável *$oArt*, que só terá o seu conteúdo alterado quando algum dado for informado para a variável *$Art*, isto é, quando não for teclado um simples *ENTER* para manter o artista anterior.

O que foi visto até agora sobre o *while* foi muito pouco. Esse comando é muito utilizado, principalmente para leitura de arquivos, porém ainda nos falta bagagem para prosseguir. Depois que aprendermos mais sobre isso, veremos essa instrução mais a fundo.

O comando *until*

Este comando funciona de forma idêntica ao *while*, porém ao contrário. Disse tudo mas não disse nada, né? É o seguinte: ambos testam comandos; ambos possuem a mesma sintaxe e ambos atuam em loop; porém, o *while* executa o bloco de instruções do loop enquanto um comando for bem sucedido; já o until executa o bloco do loop até que o comando seja bem sucedido. Parece pouca coisa, mas a diferença é fundamental. A sintaxe do comando é praticamente a mesma do while. Veja:

```
until comando
do
    cmd1
    cmd2
    ...
    cmdn
done
```

e dessa forma o bloco de comandos formado pelas instruções *cmd1, cmd2,...* e *cmdn* é executado até que a execução da instrução *comando* seja bem sucedida.

Como eu te disse, *while* e *until* funcionam de forma antagônica, e isso é muito fácil de demonstrar: em uma guerra, sempre que se inventa uma arma, o inimigo busca uma solução para neutralizá-la. Foi baseado nesse principio belicoso que meu chefe desenvolveu, no mesmo servidor em que eu executava o *logaute.sh*, um script para controlar o meu horário de chegada.

Um dia tivemos um problema na rede. Ele me pediu para dar uma olhada no micro dele e me deixou sozinho na sala. Resolvi bisbilhotar os arquivos – guerra é guerra – e veja só o que descobri:

```
$cat chegada.sh
#!/bin/bash
until who | grep julio
do
    sleep 30
done
echo $(date "+ Em %d/%m às ➥
%H:%Mh") > relapso.log
```

Olha que safado! O cara estava montando um log com os meus horários de chegada, e ainda por cima chamou o arquivo de *relapso.log*! O que será que ele quis dizer com isso?

Nesse script, o pipeline *who | grep julio*, será bem sucedido somente quando *julio* for encontrado na saída do comando *who*, isto é, quando eu me "logar" no servidor. Até que isso aconteça, o comando *sleep*, que forma o bloco de instruções do *until*, colocará o programa em espera por 30 segundos. Quando esse loop encerrar-se, será enviada uma mensagem para o arquivo *relapso.log*. Supondo que no dia 20/01 eu me "loguei" às 11:23 horas, a mensagem seria a seguinte:

Listagem 1

```
$ cat musinc.sh
#!/bin/bash
# Cadastra CDs (versao 4)
#
clear
read -p "Título do Álbum: " Tit
[ "$Tit" ] || exit 1 # Fim da execução se título vazio
if grep "^$Tit\^" musicas > /dev/null
then
    echo "Este álbum já está cadastrado"
    exit 1
fi
Reg="$Tit^"
Cont=1
oArt=
while true
do
    echo "Dados da trilha $Cont:"
    read -p "Música: " Mus
    [ "$Mus" ] || break       # Sai se vazio
    read -p "Artista: $oArt // " Art
    [ "$Art" ] && oArt="$Art" # Se vazio Art anterior
    Reg="$Reg$oArt~$Mus:"     # Montando registro
    Cont=$((Cont + 1))
    # A linha anterior tb poderia ser ((Cont++))
done
echo "$Reg" >> musicas
sort musicas -o musicas
```

Em 20/01 às 11:23h

Voltando à nossa CDteca, quando vamos cadastrar músicas seria ideal que pudéssemos cadastrar diversos CDs de uma vez só. Na última versão do programa isso não ocorre: a cada CD cadastrado o programa termina. Veja na listagem 2 como melhorá-lo.

Nesta versão, um loop maior foi adicionado antes da leitura do título, que só terminará quando a variável *$Para* deixar de ser vazia. Caso o título do álbum não seja informado, a variável *$Para* receberá um valor (coloquei 1, mas poderia ter colocado qualquer coisa) para sair desse loop, terminando o programa. No resto, o script é idêntico à versão anterior.

Atalhos no loop

Nem sempre um ciclo de programa, compreendido entre um *do* e um *done*, sai pela porta da frente. Em algumas oportunidades, temos que colocar um comando que aborte de forma controlada esse loop. De maneira inversa, algumas vezes desejamos que o fluxo de execução do programa volte antes de chegar ao *done*. Para isso, temos respectivamente os comandos *break* (que já vimos rapidamente nos exemplos do comando *while*) e *continue*, que funcionam da forma mostrada na figura 1.

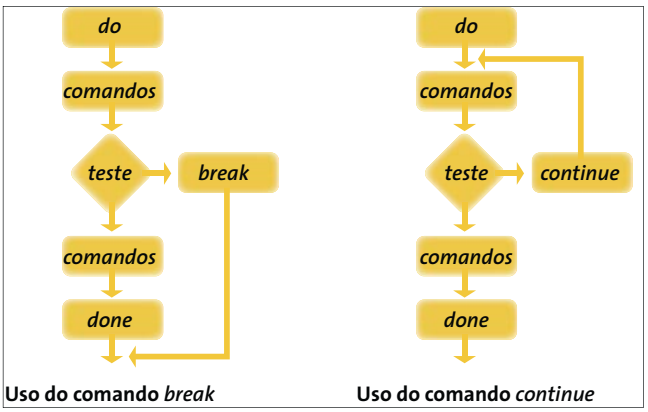


Figura 1: A estrutura dos comandos *break* e *continue*, usados para controlar o fluxo de execução em loops.

O que eu não havia dito anteriormente é que nas suas sintaxes genéricas eles aparecem da seguinte forma:

```
break [qtd loop]
```

e também:

```
continue [qtd loop]
```

Onde *qtd loop* representa a quantidade dos loops mais internos sobre os quais os comandos irão atuar. Seu valor por *default* é 1.

Duvido que você nunca tenha apagado um arquivo e logo após deu um tabefe na testa se xingando porque não devia tê-lo removido. Pois é, na décima vez que fiz esta besteira, criei um script para simular uma lixeira, isto é, quando mando remover um (ou vários) arquivo(s), o programa "finge" que deletou, mas no duro o que ele fez foi mandá-lo(s) para o diretório /tmp/LoginName_do_usuario. Chamei esse programa de *erreeme* e no arquivo */etc/profile* coloquei a seguinte linha, que cria um "apelido" para ele:

```
alias rm=erreeme
```

Veja o programa na listagem 3. Como você pode ver, a maior parte do script é formada por pequenas críticas aos parâmetros informados, mas como o script pode ter recebido diversos arquivos a remover, a cada arquivo que não se encaixa dentro do especificado há

Listagem 2

```
$ cat musinc.sh
#!/bin/bash
# Cadastra CDs (versao 5)
#
Para=
until [ "$Para" ]
do
    clear
    read -p "Título do Álbum: " Tit
    if  [ ! "$Tit" ] # Se titulo vazio...
    then
        Para=1     # Liguei flag de saída
    else
        if grep "^$Tit\^" musicas > /dev/null
        then
            echo "Este álbum já está cadastrado"
            exit 1
        fi
        Reg="$Tit^"
        Cont=1
        oArt=
        while [ "$Tit" ]
        do
            echo Dados da trilha $Cont:
            read -p "Música: " Mus
            [ "$Mus" ] || break       # Sai se vazio
            read -p "Artista: $oArt // " Art
            [ "$Art" ] && oArt="$Art" # Se vazio Art anterior
            Reg="$Reg$oArt~$Mus:"     # Montando registro
            Cont=$((Cont + 1))
            # A linha anterior tb poderia ser ((Cont++))
        done
        echo "$Reg" >> musicas
        sort musicas -o musicas
    fi
done
```

Listagem 3: erreeme.sh

```
$ cat erreeme.sh
#!/bin/bash
#
# Salvando cópia de um arquivo antes de removê-lo

# Tem de ter um ou mais arquivos a remover
if  [ $# -eq 0 ]
then
    echo "Erro -> Uso: erreeme arq [arq] ... [arq]"
    echo "O uso de metacaracteres e' permitido. Ex.➥
erreeme arq*"
    exit 1
fi

# Variável do sistema que contém o nome do usuário.
MeuDir="/tmp/$LOGNAME"
# Se não existir o meu diretório sob o /tmp...
if  [ ! -d $MeuDir ]
then
    mkdir $MeuDir      # Vou criá-lo
fi

# Se não posso gravar no diretório...
if  [ ! -w $MeuDir ]
then
    echo "Impossivel salvar arquivos em $MeuDir. ➥
Mude as permissões..."
    exit 2
fi

# Variável que indica o cod. de retorno do programa
Erro=0
# Um for sem o in recebe os parametros passados
for Arq
do
# Se este arquivo não existir...
    if  [ ! -f $Arq ]
    then
        echo "$Arq nao existe."
        Erro=3
        continue       # Volta para o comando for
    fi

# Cmd. dirname informa nome do dir de $Arq
    DirOrig=`dirname $Arq`
# Verifica permissão de gravacao no diretório
    if  [ ! -w $DirOrig ]
    then
        echo "Sem permissão no diretorio de $Arq"
        Erro=4
        continue         # Volta para o comando for
    fi

# Se estou "esvaziando a lixeira"...
    if  [ "$DirOrig" = "$MeuDir" ]
    then
        echo "$Arq ficara sem copia de seguranca"
        rm -i $Arq      # Pergunta antes de remover
        # Será que o usuário removeu?
        [ -f $Arq ] || echo "$Arquivo removido"
        continue
    fi

# Guardo no fim do arquivo o seu diretório original➥
para usá-lo em um script de undelete
    cd $DirOrig
    pwd >> $Arq
    mv $Arq $MeuDir  # Salvo e removo
    echo "$Arq removido"
done

# Passo eventual número do erro para o código
# de retorno
exit $Erro
```

um *continue*, para que a seqüência volte para o loop do *for* de forma a receber outros arquivos.

Quando você está no Windows (com perdão da má palavra) e tenta remover aquele monte de lixo com nomes esquisitos como HD04TG.TMP, se der erro em um dos arquivos os outros não são removidos, não é? Então, o *continue* foi usado para evitar que uma impropriedade dessas ocorra, isto é, mesmo que dê erro na remoção de um arquivo, o programa continuará removendo os outros que foram passados.

– Eu acho que a esta altura você deve estar curioso para ver o programa que restaura o arquivo removido, não é? Pois então aí vai vai um desafio: faça-o em casa e me traga para discutirmos no nosso próximo encontro aqui no boteco.
– Poxa, mas nesse eu acho que vou dançar, pois não sei nem como começar...
– Cara, este programa é como tudo o que se faz em Shell: extremamente fácil. É para ser feito em, no máximo, 10 linhas. Não se esqueça de que o arquivo está salvo em */tmp/$LOGNAME* e que sua última linha é o diretório em que ele residia antes de ser "removido". Também não se esqueça de criticar se foi passado o nome do arquivo a ser removido.
– É eu vou tentar, mas sei não...
– Tenha fé, irmão, eu tô te falando que é mole! Qualquer dúvida é só passar um email para julio.neves@gmail.com. Agora chega de papo que eu já estou de goela seca de tanto falar. Me acompanha no próximo chope ou já vai sair correndo para fazer o script que passei?
– Deixa eu pensar um pouco...
– Chico, traz mais um chope enquanto ele pensa!

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


Dave Hamilton - www.sxc.hu

Curso de Shell Script

Papo de Botequim
Parte VII

De pouco adianta ter acesso à informação se ela não puder ser apresentada de forma atraente e que facilite a compreensão. O comando *tput* pode ser usado por shell scripts para posicionar caracteres e criar todos os tipos de efeito com o texto mostrado na tela. Garçom, solta uma geladinha!

por Julio Cezar Neves

Cumequié, rapaz! Derreteu os pensamentos para fazer o scriptzinho que eu te pedi?

– É, eu realmente tive de colocar muita pensação na tela preta, mas acho que finalmente consegui! Bem, pelo menos nos testes que fiz a coisa funcionou, mas você tem sempre que botar chifres em cabeça de cachorro!

– Não é bem assim. É que programar em Shell Script é muito fácil, mas o que é realmente importante são as dicas e macetes que não são triviais. As correções que faço são justamente para mostrá-los. Mas vamos pedir dois chopes enquanto dou uma olhadela no teu script lá na **listagem 1**. Aê Chico, traz dois chopes! E não se esqueça que um deles é sem colarinho!

– Peraí, deixa eu ver se entendi o que você fez: você coloca na variável `Dir` a última linha do arquivo a ser restaurado, em nosso caso `/tmp/$LOGNAME/$1` (onde `$LOGNAME` é o nome do usuário logado, e `$1` é o primeiro parâmetro que você passou ao script), já que foi lá que armazenamos o nome e caminho originais do arquivo antes de movê-lo para o diretório (definido na variável `Dir`). O comando `grep -v` apaga essa linha, restaurando o arquivo ao estado original, e o manda de volta pra onde ele veio. A última linha o apaga da "lixeira". Sensacional! Impecável! Nenhum erro! Viu? Você já está pegando as manhas do shell!

– Então vamos lá, chega de lesco-lesco e blá-blá-blá, sobre o quê nós vamos falar hoje?

– É, tô vendo que o bichinho do shell te pegou. Vamos ver como ler dados, mas antes vou te mostrar um comando que te dá todas as ferramentas para formatar uma tela de entrada de dados.

Listagem 1 – restaura.sh

```
01 #!/bin/bash
02 #
03 # Restaura arquivos deletados via erreeme
04 #
05
06 if [ $# -eq 0 ]
07 then
08     echo "Uso: $0 <Nome do arquivo a ser restaurado>"
09     exit 1
10 fi
11 # Pega nome do arquivo/diretório original na última linha
12 Dir='tail -1 /tmp/$LOGNAME/$1'
13 # O grep -v exclui a última linha e recria o arquivo com o diretório
14 # e nome originalmente usados
15 grep -v $Dir /tmp/$LOGNAME/$1 > $Dir/$1
16 # Remove o arquivo que já estava moribundo
17 rm /tmp/$LOGNAME/$1
```

O comando tput

O principal uso desse comando é o posicionamento do cursor na tela. Alguns parâmetros podem não funcionar se o modelo de terminal definido pela variável de ambiente `$TERM` não suportá-los. A **tabela 1** apresenta apenas os principais parâmetros e os efeitos resultantes, mas existem muito mais deles. Para saber tudo sobre o *tput*, veja a referência [1].

Vamos fazer um programa bem besta e fácil para ilustrar melhor o uso desse comando. É uma versão do famigerado "Alô Mundo", só que dessa vez a frase será escrita no centro da tela e em vídeo reverso. Depois disso, o cursor voltará para a posição original. Veja a **listagem 2**.

Como o programa já está todo comentado, acho que a única linha que precisa de explicação é a 8, onde criamos a variável `Coluna`. O estranho ali é aquele número 9, que na verdade indica o tamanho da cadeia de caracteres que vou escrever na tela. Dessa forma, este programa somente conseguiria centralizar cadeias de 9 caracteres, mas veja isto:

```
$ var=Papo
$ echo ${#var}
4
$ var="Papo de Botequim"
$ echo ${#var}
16
```

Tabela 1: Parâmetros do tput

Parâmetro	Efeito
`cup lin col`	***CU**rsor **P**osition* – Posiciona o cursor na linha *lin* e coluna *col*. A origem (0,0) fica no canto superior esquerdo da tela.
`bold`	Coloca a tela em modo negrito
`rev`	Coloca a tela em modo de vídeo reverso
`smso`	Idêntico ao anterior
`smul`	Sublinha os caracteres
`blink`	Deixa os caracteres piscando
`sgr0`	Restaura a tela a seu modo normal
`reset`	Limpa o terminal e restaura suas definições de acordo com *terminfo*, ou seja, o terminal volta ao comportamento padrão definido pela variável de ambiente `$TERM`
`lines`	Informa a quantidade de linhas que compõem a tela
`cols`	Informa a quantidade de colunas que compõem a tela
`el`	***E**rase **L**ine* – Apaga a linha a partir da posição do cursor
`ed`	***E**rase **D**isplay* – Apaga a tela a partir da posição do cursor
`il n`	***I**nsert **L**ines* – Insere n linhas a partir da posição do cursor
`dl n`	***D**elete **L**ines* – Remove n linhas a partir da posição do cursor
`dch n`	***D**elete **CH**aracters* – Apaga n caracteres a partir da posição do cursor
`sc`	***S**ave **C**ursor position* – Salva a posição do cursor
`rc`	***R**estore **C**ursor position* – Coloca o cursor na posição marcada pelo último `sc`

Ahhh, melhorou! Então agora sabemos que a construção `${#variavel}` devolve a quantidade de caracteres da variável. Assim sendo, vamos otimizar o nosso programa para que ele escreva em vídeo reverso, no centro da tela (e independente do número de caracteres) a cadeia de caracteres passada como parâmetro e depois retorne o cursor à posição em que estava antes da execução do script. Veja o resultado na **listagem 3**.

Este script é igual ao anterior, só que trocamos o valor fixo na variável Coluna (9) por `${#1}`, onde esse `1` é `$1`, ou seja, essa construção devolve o número de caracteres do primeiro parâmetro passado para o programa. Se o parâmetro tivesse espaços em branco, seria preciso colocá-lo entre aspas, senão o `$1` levaria em conta somente o pedaço antes do primeiro espaço. Para evitar este aborrecimento, é só substituir o `$1` por `$*`, que como sabemos é o conjunto de todos os parâmetros. Então a linha 8 ficaria assim:

```
# Centralizando a mensagem na tela
Coluna=`$(((Colunas - ${#*}) / 2))`
```

e a linha 12 (`echo $1`) passaria a ser:

```
echo $*
```

Lendo dados da tela

Bem, a partir de agora vamos aprender tudo sobre leitura. Só não posso ensinar a ler cartas e búzios porque se soubesse estaria rico, num *pub* Londrino tomando um *scotch* e não em um boteco tomando chope. Mas vamos em frente.

Da última vez em que nos encontramos eu dei uma palhinha sobre o comando `read`. Antes de entrarmos em detalhes, veja só isso:

```
$ read var1 var2 var3
Papo de Botequim
$ echo $var1
Papo
$ echo $var2
de
$ echo $var3
Botequim
$ read var1 var2
Papo de Botequim
$ echo $var1
Papo
$ echo $var2
de Botequim
```

Como você viu, o *read* recebe uma lista de parâmetros separada por espaços em branco e coloca cada item dessa lista em uma variável. Se a quantidade de variáveis for menor que a quantidade de itens, a última variável recebe o restante deles. Eu disse lista separada por espaços em branco, mas agora que você já conhece tudo sobre o `$IFS` (*Inter Field Separator* – Separador entre campos), que

Listagem 2: alo.sh

```
01 #!/bin/bash
02 # Script bobo para testar
03 # o comando tput (versao 1)
04
05 Colunas=`tput cols`     #  Salva a quantidade de colunas na tela
06 Linhas=`tput lines`     #  Salva a quantidade linhas na tela
07 Linha=$((Linhas / 2))   # Qual é a linha central da tela?
08 Coluna=$(((Colunas - 9) / 2)) # Centraliza a mensagem na tela
09 tput sc                # Salva a posição do cursor
10 tput cup $Linha $Coluna #  Posiciona o cursor antes de escrever
11 tput rev               #  Vídeo reverso
12 echo Alô Mundo
13 tput sgr0              #  Restaura o vídeo ao normal
14 tput rc                #  Restaura o cursor à posição original
```

Listagem 3: alo.sh melhorado

```
01 #!/bin/bash
02 # Script bobo para testar
03 # o comando tput (versão 2.0)
04
05 Colunas=`tput cols`     #  Salva a quantidade de colunas na tela
06 Linhas=`tput lines`     #  Salva a quantidade de linhas na tela
07 Linha=$((Linhas / 2))   # Qual é a linha central da tela?
08 Coluna=$(((Colunas - ${#1}) / 2)) # Centraliza a mensagem na tela
09 tput sc                # Salva a posicao do cursor
10 tput cup $Linha $Coluna #  Posiciona o cursor antes de escrever
11 tput rev               #  Vídeo reverso
12 echo $1
13 tput sgr0              #  Restaura o vídeo ao normal
14 tput rc                #  Devolve o cursor à posição original
```

eu te apresentei quando falávamos do comando `for`, será que ainda acredita nisso? Vamos testar:

```
$ oIFS="$IFS"
$ IFS=:
$ read var1 var2 var3
Papo de Botequim
$ echo $var1
Papo de Botequim
$ echo $var2
$ echo $var3
$ read var1 var2 var3
Papo:de:Botequim
$ echo $var1
Papo
$ echo $var2
de
$ echo $var3
Botequim
$ IFS="$oIFS"
```

Viu? eu estava furado! O `read` lê uma lista, assim como o `for`, separada pelos caracteres da variável `$IFS`. Veja como isso pode facilitar a sua vida:

```
$ grep julio /etc/passwd
julio:x:500:544:Julio C. Neves - 7070:➥
/home/julio:/bin/bash
$ oIFS="$IFS"    #  Salva o IFS antigo.
$ IFS=:
$ grep julio /etc/passwd | read lname ➥
lixo uid gid coment home shell
$ echo -e "$lname\n$uid\n$gid\n$coment➥
\n$home\n$shell"
julio
500
544
Julio C. Neves - 7070
/home/julio
/bin/bash
$ IFS="$oIFS"   #  Restaura o IFS
```

Como você viu, a saída do `grep` foi redirecionada para o comando `read`, que leu todos os campos de uma só tacada. A opção `-e` do `echo` foi usada para que o `\n` fosse entendido como uma quebra de linha (*new line*) e não como um literal. Sob o Bash existem diversas opções do `read` que servem para facilitar a sua vida. Veja a **tabela 2**.

E agora direto aos exemplos curtos para demonstrar estas opções. Para ler um campo "Matrícula":

```
# -n não salta linha
$ echo -n "Matricula: "; read Mat
Matricula: 12345
$ echo $Mat
12345
```

Podemos simplificar as coisas usando a opção `-p`:

```
$ read -p "Matricula: " Mat
Matricula: 12345
$ echo $Mat
12345
```

E podemos ler apenas uma quantidade pré-determinada de caracteres:

```
$ read -n5 -p"CEP: " Num ; read -n3 ➥
-p- Compl
CEP: 12345-678$
$ echo $Num
12345
$ echo $Compl
678
```

No exemplo acima executamos duas vezes o comando `read`: um para a primeira parte do CEP e outra para o seu complemento, deste modo formatando a entrada de dados. O cifrão (`$`) logo após o último algarismo digitado é necessário porque o `read` não inclui por padrão um caractere *new line* implícito, como o echo.

Para ler só durante um determinado limite de tempo (também conhecido como *time out*):

```
$ read -t2 -p "Digite seu nome completo: ➥
" Nom || echo 'Eita moleza!'
Digite seu nome completo: Eita moleza!
$ echo $Nom
```

O exemplo acima foi uma brincadeira, pois eu só tinha 2 segundos para digitar o meu nome completo e mal tive tempo de teclar um *J* (aquele colado no *Eita*), mas ele serviu para mostrar duas coisas:

- **1)** O comando após o par de barras verticais (o *ou* – *or* – lógico, lembra-se?) será executado caso a digitação não tenha sido concluída no tempo estipulado;
- **2)** A variável `Nom` permaneceu vazia. Ela só receberá um valor quando o ***ENTER*** for teclado.

```
$ read -sp "Senha: "
Senha: $ echo $REPLY
segredo :)
```

Aproveitei um erro no exemplo anterior para mostrar um macete. Quando escrevi a primeira linha, esqueci de colocar o nome da variável que iria receber a senha e só notei isso quando ia escrevê-la. Felizmente a variável `$REPLY` do Bash contém a última seqüência de caracteres digitada – e me aproveitei disso para não perder a viagem. Teste você mesmo o que acabei de fazer.

O exemplo que dei, na verdade, era para mostrar que a opção `-s` impede que o que está sendo digitado seja mostrado na tela. Como no exemplo anterior, a falta do *new line* fez com que o prompt de comando (`$`) permanecesse na mesma linha.

Agora que sabemos ler da tela, vejamos como se lêem os dados dos arquivos.

Lendo arquivos

Como eu já havia lhe dito, e você deve se lembrar, o `while` testa um comando e executa um bloco de instruções enquanto esse comando for bem sucedido. Ora, quando você está lendo um arquivo para o qual você tem permissão de leitura, o `read` só será mal sucedido quando alcançar o `EOF` (*End Of File* – Fim do Arquivo). Portanto, podemos ler um arquivo de duas maneiras. A primeira é redirecionando a entrada do arquivo para o bloco `while`, assim:

```
while read Linha
do
    echo $Linha
done < arquivo
```

Tabela 2: Opções do read

Opção	Ação
`-p prompt`	Escreve "prompt" na tela antes de fazer a leitura
`-n num`	Lê até `num` caracteres
`-t seg`	Espera `seg` segundos para que a leitura seja concluída
`-s`	Não exibe na tela os caracteres digitados.

A segunda é redirecionando a saída de um `cat` para o `while`, da seguinte maneira:

```
cat arquivo |
while read Linha
do
    echo $Linha
done
```

Cada um dos processos tem suas vantagens e desvantagens. O primeiro é mais rápido e não necessita de um subshell para assisti-lo mas, em contrapartida, o redirecionamento fica pouco visível em um bloco de instruções grande, o que por vezes prejudica a visualização do código. O segundo processo traz a vantagem de que, como o nome do arquivo está antes do while, a visualização do código é mais fácil. Entretanto, o Pipe (|) chama um subshell para interpretá-lo, tornando o processo mais lento e pesado. Para ilustrar o que foi dito, veja os exemplos a seguir:

```
$ cat readpipe.sh
#!/bin/bash
# readpipe.sh
# Exemplo de read passando um arquivo
# por um pipe.
Ultimo="(vazio)"
# Passa o script ($0) para o while
cat $0 | while read Linha
do
    Ultimo="$Linha"
    echo "-$Ultimo-"
done
echo "Acabou, Último=:$Ultimo:"
```

Vamos ver o resultado de sua execução:

```
$ readpipe.sh
-#!/bin/bash-
-# readpipe.sh-
-# Exemplo de read passando um arquivo
-# por um pipe.-
--
-Ultimo="(vazio)"-
-# Passa o script ($0) para o while-
-cat $0 | -
-while read Linha-
-do-
-Ultimo="$Linha"-
-echo "-$Ultimo-"-
-done-
-echo "Acabou, Último=:$Ultimo:"-
 Acabou, Último=:(vazio):
```

Como você viu, o script lista suas próprias linhas com um sinal de menos (-) antes e outro depois de cada uma e, no final, exibe o conteúdo da variável `$Ultimo`. Repare, no entanto, que o conteúdo dessa variável permanece vazio. Ué, será que a variável não foi atualizada? Foi, e isso pode ser comprovado porque a linha `echo "-$Ultimo-"` lista corretamente as linhas. Então por que isso aconteceu?

Como eu disse, o bloco de instruções redirecionado pelo pipe (|) é executado em um subshell e, lá, as variáveis são atualizadas. Quando esse subshell termina, as atualizações das variáveis vão para as profundezas do inferno junto com ele. Repare que vou fazer uma pequena mudança no script, passando o arquivo por redirecionamento de entrada (<), e as coisas passarão a funcionar na mais perfeita ordem:

```
$ cat redirread.sh
#!/bin/bash
# redirread.sh
# Exemplo de read passando o arquivo
# por um pipe.
Ultimo="(vazio)"
# Passa o script ($0) para o while
while read Linha
do
    Ultimo="$Linha"
    echo "-$Ultimo-"
done < $0
echo "Acabou, Último=:$Ultimo:"
```

Veja como ele roda perfeitamente:

```
$ redirread.sh
-#!/bin/bash-
-# redirread.sh-
-# Exemplo de read passando o arquivo
-# por um pipe.-
--
-Ultimo="(vazio)"-
-while read Linha-
-do-
-Ultimo="$Linha"-
-echo "-$Ultimo-"-
-# Passa o script ($0) para o while-
-done < $0-
-echo "Acabou, Último=:$Ultimo:"-
 Acabou, Último=:echo "Acabou,➥
 Último=:$Ultimo:":
```

Bem, amigos da Rede Shell, para finalizar a aula sobre o comando `read` só falta mais um pequeno e importante macete que vou mostrar com um exemplo prático. Suponha que você queira listar um arquivo e quer que a cada dez registros essa listagem pare para que o operador possa ler o conteúdo da tela, e que ela só continue depois de o operador pressionar qualquer tecla. Para não gastar papel (da Linux Magazine), vou fazer essa listagem na horizontal. Meu arquivo (`numeros`) tem 30 registros com números seqüenciais. Veja:

```
$ seq 30 > numeros
$ cat 10porpag.sh
#!/bin/bash
# Programa de teste para escrever
# 10 linhas e parar para ler
# Versão 1
while read Num
do
    let ContLin++   # Contando...
    # -n para não saltar linha
    echo -n "$Num "
    ((ContLin % 10)) > /dev/null || read
done < numeros
```

Na tentativa de fazer um programa genérico criamos a variável `$ContLin` (na vida real, os registros não são somente números seqüenciais) e, quando testamos se o resto da divisão era zero, mandamos a saída para `/dev/null`, pra que ela não apareça na tela. Mas quando fui executar o programa descobri a seguinte zebra:

```
$ 10porpag.sh
1 2 3 4 5 6 7 8 9 10 12 13 14 15 16 17 ➥
18 19 20 21 23 24 25 26 27 28 29 30
```

Repare que faltou o número 11 e a listagem não parou no `read`. Toda a entrada do loop estava redirecionada para o arquivo `numeros` e a leitura foi feita em cima desse arquivo, perdendo o número 11. Vamos mostrar como deveria ficar o código para que ele passe a funcionar a contento:

```
$ cat 10porpag.sh
#!/bin/bash
# Programa de teste para escrever
# 10 linhas e parar para ler - Versão 2
while read Num
do
    let ContLin++   # Contando...
    # -n para não saltar linha
    echo -n "$Num "
    ((ContLin % 10)) > /dev/null || read ➥
< /dev/tty
done < numeros
```

Repare que agora a entrada do `read` foi redirecionada para `/dev/tty`, que nada mais é senão o terminal corrente, explicitando desta forma que aquela leitura seria feita do teclado e não do arquivo `numeros`. É bom realçar que isso não acontece somente quando usamos o redirecionamento de entrada; se tivéssemos usado o redirecionamento via pipe (`|`), o mesmo teria ocorrido. Veja agora a execução do script:

```
$ 10porpag.sh
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
```

Isso está quase bom, mas ainda falta um pouco para ficar excelente. Vamos melhorar o exemplo para que você o reproduza e teste (mas, antes de testar, aumente o número de registros em `numeros` ou reduza o tamanho da tela, para que haja quebra de linha).

```
$ cat 10porpag.sh
#!/bin/bash
# Programa de teste para escrever
# 10 linhas e parar para ler
# Versão 3
clear
while read Num
do
    # Contando...
    ((ContLin++))
    echo "$Num"
    ((ContLin % (`tput lines` - 3))) ||
        {
        # para ler qualquer caractere
        read -n1 -p"Tecle Algo " < /dev/tty
        # limpa a tela após a leitura
        clear
        }
done < numeros
```

A mudança substancial feita neste exemplo é com relação à quebra de página, já que ela é feita a cada quantidade-de-linhas-da-tela (`tput lines`) menos (`-`) três, isto é, se a tela tem 25 linhas, o programa listará 22 registros e parará para leitura. No comando `read` também foi feita uma alteração, inserido o parâmetro `-n1` para ler somente um caractere qualquer, não necessariamente um ***ENTER***, e a opção `-p` para exibir uma mensagem.

– Bem meu amigo, por hoje é só porque acho que você já está de saco cheio...
– Num tô não, pode continuar...
– Se você não estiver eu estou... Mas já que você está tão empolgado com o shell, vou te deixar um serviço bastante simples para você melhorar a sua cdteca: Monte toda a tela com um único `echo` e depois posicione o cursor à frente de cada campo para receber o valor que será digitado pelo operador.

Não se esqueçam que, em caso de qualquer dúvida ou falta de companhia para um chope é só mandar um e-mail para julio.neves@gmail.com. Vou aproveitar também para fazer uma propaganda: digam aos amigos que quem estiver a fim de fazer um curso "porreta" de programação em shell deve mandar um e-mail para julio.neves@tecnohall.com.br para informar-se. Até mais!

Informações

[1] Página oficial do Tput: http://www.cs.utah.edu/dept/old/texinfo/tput/tput.html#SEC4

[2] Página oficial do Bash: http://www.gnu.org/software/bash/bash.html


Dave Hamilton - www.sxc.hu

Curso de Shell Script

Papo de Botequim
Parte VIII

Chegou a hora de fazer como Jack e dividir os programas em pedacinhos. Com funções e chamadas externas os scripts ficam menores, a manutenção mais fácil e ainda por cima reaproveitamos código.

por Júlio Cezar Neves

– Eaê, cara, tudo bem?

– Tudo beleza! Eu queria te mostrar o que fiz mas já sei que você vai querer molhar o bico primeiro, né?

– Só pra te contrariar, hoje não quero. Vai, mostra logo aí o que você fez.

– Poxa, o exercício que você passou é muito grande. Dá uma olhada na **listagem 1** e vê como eu resolvi:

– É, o programa tá legal, tá todo estruturadinho, mas gostaria de fazer alguns poucos comentários: só para relembrar, as seguintes construções: `[ ! $Album ] &&` e `[ $Musica ] ||` representam a mesma coisa, isto é: no caso da primeira, testamos se a variável `$Album` não (`!`) tem nada dentro, então (`&&`)... Na segunda, testamos se `$Musica` tem algum dado, senão (`||`)...

Se você reclamou do tamanho do programa, é porque ainda não te dei algumas dicas. Repare que a maior parte do script é para mostrar mensagens centralizadas na penúltima linha da tela. Repare ainda que algumas mensagens pedem um `S` ou um `N` como resposta e outras são só de advertência. Isso é um caso típico que pede o uso de funções, que seriam escritas somente uma vez e executadas em diversos pontos do script. Vou montar duas funções para resolver esses casos e vamos incorporá-las ao seu programa para ver o resultado final.

– Chico! Agora traz dois chopes, um sem colarinho, para me dar inspiração. E você, de olho na **listagem 2**.

Como podemos ver, uma função é definida quando digitamos `nome_da_função ()` e todo o seu corpo está entre chaves (`{}`). Já conversamos aqui no boteco sobre passagem de parâmetros e as funções os recebem da mesma forma, isto é, são parâmetros posicionais (`$1, $2, ..., $n`). Todas as regras que se aplicam à passagem de parâmetros para programas também valem para funções, mas é muito importante realçar que os parâmetros passados para um programa não se confundem com aqueles que são passados para suas funções. Isso significa, por exemplo, que o `$1` de um script é diferente do `$1` de uma de suas funções internas.

Repare que as variáveis `$Msg, $TamMsg` e `$Col` são de uso restrito dessa rotina e, por isso, foram criadas como variáveis locais. A razão é simplesmente a economia de memória, já que ao sair da rotina elas serão devidamente detonadas, coisa que não aconteceria se eu não tivesse usado esse artifício.

A linha de código que cria a variável local `Msg` concatena ao texto recebido (`$1`) um parêntese, a resposta padrão (`$2`) em caixa alta, uma barra, a outra resposta (`$3`) em caixa baixa e finaliza fechando o parêntese. Uso essa convenção para, ao

Listagem 1: *musinc5.sh*

```
01 $ cat musinc5.sh
02 #!/bin/bash
03 # Cadastra CDs (versao 5)
04 #
05 clear
06 LinhaMesg=$((`tput lines` - 3)) # Linha onde serão mostradas as msgs para o operador
07 TotCols=$(tput cols)           # Qtd colunas da tela para enquadrar msgs
08 echo "
                                   Inclusão de Músicas
                                   ======= == =======
                  Título do Álbum:
                                        | Este campo foi
                  Faixa:               < criado somente para
                                        | orientar o preenchimento
                  Nome da Música:
                  Intérprete:"           # Tela montada com um único echo
09 while true
10 do
11     tput cup 5 38; tput el    # Posiciona e limpa linha                    ➡
```

```
12    read Album
13    [ ! "$Album" ] &&         # Operador deu <ENTER>
14        {
15            Msg="Deseja Terminar? (S/n)"
16            TamMsg=${#Msg}
17            Col=$(((TotCols - TamMsg) / 2))        # Centraliza msg na linha
18            tput cup $LinhaMesg $Col
19            echo "$Msg"
20            tput cup $LinhaMesg $((Col + TamMsg + 1))
21            read -n1 SN
22            tput cup $LinhaMesg $Col; tput el      # Apaga msg da tela
23            [ $SN = "N" -o $SN = "n" ] && continue # $SN é igual a N ou (-o) n?
24            clear; exit                            # Fim da execução
25        }
26    grep "^$Album\^" musicas > /dev/null &&
27        {
28            Msg="Este álbum já está cadastrado"
29            TamMsg=${#Msg}
30            Col=$(((TotCols - TamMsg) / 2))        # Centraliza msg na linha
31            tput cup $LinhaMesg $Col
32            echo "$Msg"
33            read -n1
34            tput cup $LinhaMesg $Col; tput el      # Apaga msg da tela
35            continue                               # Volta para ler outro álbum
36        }
37    Reg="$Album^"                        # $Reg receberá os dados para gravação
38    oArtista=                            # Variável que guarda artista anterior
39    while true
40    do
41        ((Faixa++))
42        tput cup 7 38
43        echo $Faixa
44        tput cup 9 38                    # Posiciona para ler música
45        read Musica
46        [ "$Musica" ] ||                 # Se o operador tiver dado <ENTER>...
47            {
48               Msg="Fim de Álbum? (S/n)"
49               TamMsg=${#Msg}
50               Col=$(((TotCols - TamMsg) / 2))        # Centraliza msg na linha
51               tput cup $LinhaMesg $Col
52               echo "$Msg"
53               tput cup $LinhaMesg $((Col + TamMsg + 1)
54               read -n1 SN
55               tput cup $LinhaMesg $Col; tput el      # Apaga msg da tela
56               [ "$SN" = N -o "$SN" = n ] && continue # $SN é igual a N ou (-o) n?
57               break                                  # Sai do loop para gravar
58            }
59        tput cup 11 38                               # Posiciona para ler Artista
60        [ "$oArtista" ] && echo -n "($oArtista) "  # Artista anterior é default
61        read Artista
62        [ "$Artista" ] && oArtista="$Artista"
63        Reg="$Reg$oArtista~$Musica:"         # Montando registro
64        tput cup 9 38; tput el               # Apaga Música da tela
65        tput cup 11 38; tput el              # Apaga Artista da tela
66    done
67    echo "$Reg" >> musicas                   # Grava registro no fim do arquivo
38    sort musicas -O musicas                  # Classifica o arquivo
69 done
```

mesmo tempo, mostrar as opções disponíveis e realçar a resposta oferecida como padrão.

Quase no fim da rotina, a resposta recebida (`$SN`) é convertida para caixa baixa (minúsculas) de forma que no corpo do programa não precisemos fazer esse teste. Veja na **listagem 3** como ficaria a função para exibir uma mensagem na tela.

Essa é uma outra forma de definir uma função: não a chamamos, como no exemplo anterior, usando uma construção com a sintaxe `nome_da_função ()`, mas sim como `function nome_da_função`. Em nada mais ela difere da anterior, exceto que, como consta dos comentários, usamos a variável `$*` que, como já sabemos, representa o conjunto de todos os parâmetros passados ao script, para que o programador não precise usar aspas envolvendo a mensagem que deseja passar à função.

Para terminar com esse blá-blá-blá, vamos ver na **listagem 4** as alterações no programa quando usamos o conceito de funções:

Repare que a estrutura do script segue a ordem *Variáveis Globais*, *Funções* e *Corpo do Programa*. Esta estruturação se deve ao fato de Shell Script ser uma linguagem interpretada, em que o programa é lido da esquerda para a direita e de cima para baixo. Para ser vista pelo script e suas funções, uma variável deve ser declarada (ou inicializada, como preferem alguns) antes de qualquer outra coisa. Por sua vez, as funções devem ser declaradas antes do corpo do programa propriamente dito. A explicação é simples: o interpretador de comandos do shell deve saber do que se trata a função antes que ela seja chamada no programa principal.

Uma coisa bacana na criação de funções é fazê-las tão genéricas quanto possível, de forma que possam ser reutilizadas em outros scripts e aplicativos sem a necessidade de reinventarmos a roda. As duas funções que acabamos de ver são bons exemplos, pois é difícil um script de entrada de dados que não use uma rotina como a `MandaMsg` ou que não interaja com o operador por meio de algo semelhante à `Pergunta`.

Conselho de amigo: crie um arquivo e anexe a ele cada função nova que você criar. Ao final de algum tempo você terá uma bela biblioteca de funções que lhe poupará muito tempo de programação.

O comando source

Veja se você nota algo de diferente na saída do *ls* a seguir:

```
$ ls -la .bash_profile
-rw-r--r--  1 Julio  unknown   4511 Mar 18 17:45 .bash_profile
```

Não olhe a resposta não, volte a prestar atenção! Bem, já que você está mesmo sem saco de pensar e prefere ler a resposta, vou te dar uma dica: acho que você já sabe que o `.bash_profile` é um dos scripts que são automaticamente executados quando você se "loga" (ARRGGHH! Odeio esse termo!) no sistema. Agora olhe novamente para a saída do comando *ls* e me diga o que há de diferente nela.

Como eu disse, o `.bash_profile` é executado durante o *logon*, mas repare que ele não tem nenhuma permissão de execução. Isso acontece porque se você o executasse como qualquer outro script careta, no fim de sua execução todo o ambiente por ele gerado morreria junto com o shell sob o qual ele foi executado (você se lembra de que todos os scripts são executados em *sub-shells*, né?). Pois é para coisas assim que existe o comando *source*, também conhecido por "." (ponto). Este comando faz com que o script que lhe for passado como parâmetro não seja executado em um *sub-shell*. Mas é melhor um exemplo que uma explicação em 453 palavras. Veja o scriptzinho a seguir:

```
$ cat script_bobo
cd ..
ls
```

Ele simplesmente deveria ir para o diretório acima do diretório atual. Vamos executar uns comandos envolvendo o `script_bobo` e analisar os resultados:

```
$ pwd
/home/jneves
$ script_bobo
jneves    juliana   paula     silvie
$ pwd
/home/jneves
```

Se eu mandei ele subir um diretório, por que não subiu? Opa, peraí que subiu sim! O sub-shell que foi criado para executar o script tanto subiu que listou os diretórios dos quatro usuários abaixo do diretório `/home`. Só que assim que a execução do script terminou, o sub-shell foi para o beleléu e, com ele, todo o ambiente criado. Agora preste atenção no exemplo abaixo e veja como a coisa muda de figura:

```
$ source script_bobo
jneves    juliana   paula     silvie
$ pwd
/home
$ cd -
/home/jneves
$ . script_bobo
jneves    juliana   paula     silvie
$ pwd
/home
```

Listagem 2: Função *Pergunta*

```
01 Pergunta ()
02     {
03       #  A função recebe 3 parâmetros na seguinte ordem:
04       #  $1 - Mensagem a ser mostrada na tela
05       #  $2 - Valor a ser aceito com resposta padrão
06       #  $3 - O outro valor aceito
07       #  Supondo que $1=Aceita?, $2=s e $3=n, a linha a
08       #  seguir colocaria em Msg o valor "Aceita? (S/n)"
09       local Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
10       local TamMsg=${#Msg}
11       local Col=$(((TotCols - TamMsg) / 2))  # Centra msg na linha
12       tput cup $LinhaMesg $Col
13       echo "$Msg"
14       tput cup $LinhaMesg $((Col + TamMsg + 1))
15       read -n1 SN
16       [ ! $SN ] && SN=$2                    # Se vazia coloca default em SN
17       echo $SN | tr A-Z a-z                 # A saída de SN será em minúscula
18       tput cup $LinhaMesg $Col; tput el     # Apaga msg da tela
19       return                                # Sai da função
20     }
```

Listagem 3: Função *MandaMsg*

```
01 function MandaMsg
02     {
03          # A função recebe somente um parâmetro
04          # com a mensagem que se deseja exibir.
05          # para não obrigar o programador a passar
06          # a msg entre aspas, usaremos $* (todos
07          # os parâmetros, lembra?) e não $1.
08          local Msg="$*"
09          local TamMsg=${#Msg}
10          local Col=$(((TotCols - TamMsg) / 2)) # Centra msg na linha
11          tput cup $LinhaMesg $Col
12          echo "$Msg"
13          read -n1
14          tput cup $LinhaMesg $Col; tput el     # Apaga msg da tela
15          return                                # Sai da função
16     }
```

Listagem 4: *musinc6.sh*

```
01 $ cat musinc6
02 #!/bin/bash
03 # Cadastra CDs (versao 6)
04 #
05 # Área de variáveis globais
06 # Linha onde as mensagens serão exibidas
07 LinhaMesg=$((`tput lines` - 3))
08 # Quantidade de colunas na tela (para enquadrar as mensagens)
09 TotCols=$(tput cols)
10 # Área de funções
11 Pergunta ()
12     {
13     #  A função recebe 3 parâmetros na seguinte ordem:
14     #  $1 - Mensagem a ser dada na tela
15     #  $2 - Valor a ser aceito com resposta default
16     #  $3 - O outro valor aceito
17     #  Supondo que $1=Aceita?, $2=s e $3=n, a linha
18     #  abaixo colocaria em Msg o valor "Aceita? (S/n)"
19     local Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
20     local TamMsg=${#Msg}
21     # Centraliza a mensagem na linha
22     local Col=$(((TotCols - TamMsg) / 2))
23     tput cup $LinhaMesg $Col
24     echo "$Msg"
25     tput cup $LinhaMesg $((Col + TamMsg + 1))
26     read -n1 SN
27     [ ! $SN ] && SN=$2     # Se vazia, coloca default em SN
28     echo $SN | tr A-Z a-z  # A saída de SN será em minúsculas
29     tput cup $LinhaMesg $Col; tput el   # Apaga msg da tela
30     return                              # Sai da função
31     }
32 function MandaMsg
33     {
34     # A função recebe somente um parâmetro
35     # com a mensagem que se deseja exibir;
36     # para não obrigar o programador a passar
37     # a msg entre aspas, usaremos $* (todos
38     # os parâmetro, lembra?) e não $1.
39     local Msg="$*"
40     local TamMsg=${#Msg}
41     # Centraliza mensagem na linha
42     local Col=$(((TotCols - TamMsg) / 2))
43     tput cup $LinhaMesg $Col
44     echo "$Msg"
45     read -n1
46     tput cup $LinhaMesg $Col; tput el  # Apaga msg da tela
47     return                             # Sai da função
48     }
49 # O corpo do programa propriamente dito começa aqui
50 clear
51 # A tela a seguir é montada com um único comando echo
52 echo "
                                        Inclusão de Músicas
                                        ======== == =======
                    Título do Álbum:
                    Faixa:
                    Nome da Música:
                    Intérprete:"
53 while true
54 do
55     tput cup 5 38; tput el      # Posiciona e limpa linha
56     read Album
57     [ ! "$Album" ] &&           # Operador deu <ENTER>
58         {
59         Pergunta "Deseja Terminar" s n
60         # Agora só testo caixa baixa
61         [ $SN = "n" ] && continue
62         clear; exit             # Fim da execução
63         }
64     grep -iq "^$Album\^" musicas 2> /dev/null &&
65         {
66         MandaMsg Este álbum já está cadastrado
67         continue                # Volta para ler outro álbum
68         }
69     Reg="$Album^"       # $Reg receberá os dados de gravação
70     oArtista=           # Guardará artista anterior
71     while true
72     do
73         ((Faixa++))
74         tput cup 7 38
75         echo $Faixa
76         tput cup 9 38           # Posiciona para ler música
77         read Musica
78         [ "$Musica" ] ||   # Se o operador teclou <ENTER>...
79             {
80             Pergunta "Fim de Álbum?" s n
81             # Agora só testo a caixa baixa
82             [ "$SN" = n ] && continue
83             break           # Sai do loop para gravar dados
84             }
85         tput cup 11 38          # Posiciona para ler Artista
86         # O artista anterior é o padrão
87         [ "$oArtista" ] && echo -n "($oArtista) "
88         read Artista
89         [ "$Artista" ] && oArtista="$Artista"
90         Reg="$Reg$oArtista~$Musica:" # Montando registro
91         tput cup 9 38; tput el       # Apaga Música da tela
92         tput cup 11 38; tput el      # Apaga Artista da tela
93     done
94     # Grava registro no fim do arquivo
95     echo "$Reg" >> musicas
96     # Classifica o arquivo
97     sort musicas -o musicas
98 done
```

Listagem 5: *musinc7.sh*

```
01 $ cat musinc7.sh
02 #!/bin/bash
03 # Cadastra CDs (versao 7)
04 #
05 # Área de variáveis globais
06 LinhaMesg=$((`tput lines` - 3))  # Linha onde serão mostradas as msgs para o operador
07 TotCols=$(tput cols)            # Qtd colunas da tela para enquadrar msgs
08 # O corpo do programa propriamente dito começa aqui
09 clear
10 echo "
                                  Inclusão de Músicas
                                  ======== == =======
                    Título do Álbum:
                                       | Este campo foi
                    Faixa:             < criado somente para
                                       | orientar o preenchimento
                    Nome da Música:
                    Intérprete:"  # Tela montada com um único echo
11 while true
12 do
13     tput cup 5 38; tput el               # Posiciona e limpa linha
14     read Album
15     [ ! "$Album" ] &&                    # Operador deu <ENTER>
16         {
17         source pergunta.func "Deseja Terminar" s n
18         [ $SN = "n" ] && continue        # Agora só testo a caixa baixa
19         clear; exit                      # Fim da execução
20         }
21     grep -iq "^$Album\^" musicas 2> /dev/null &&
22         {
23         . mandamsg.func Este álbum já está cadastrado
24         continue                         # Volta para ler outro álbum
25         }
26     Reg="$Album^"                        # $Reg receberá os dados de gravação
27        oArtista=                            # Guardará artista anterior
28     while true
29     do
30         ((Faixa++))
31         tput cup 7 38
32         echo $Faixa
33         tput cup 9 38                    # Posiciona para ler música
34         read Musica
35         [ "$Musica" ] ||                 # Se o operador tiver dado <ENTER>...
36             {
37             . pergunta.func "Fim de Álbum?" s n
38             [ "$SN" = n ] && continue    # Agora só testo a caixa baixa
39             break                        # Sai do loop para gravar dados
40             }
41         tput cup 11 38                           # Posiciona para ler Artista
42         [ "$oArtista" ] && echo -n "($oArtista) "  # Artista anterior é default
43         read Artista
44         [ "$Artista" ] && oArtista="$Artista"
45         Reg="$Reg$oArtista~$Musica:"             # Montando registro
46         tput cup 9 38; tput el                   # Apaga Música da tela
47         tput cup 11 38; tput el                  # Apaga Artista da tela
48     done
49     echo "$Reg" >> musicas                 # Grava registro no fim do arquivo
50     sort musicas -o musicas              # Classifica o arquivo
51 done
```

Ahh! Agora sim! Quando passado como parâmetro do comando *source*, o script foi executado no shell corrente, deixando nele todo o ambiente criado. Agora vamos rebobinar a fita até o início da explicação sobre este comando. Lá falamos do `.bash_profile` e, a esta altura, você já deve saber que sua incumbência é, logo após o login, preparar o ambiente de trabalho para o usuário. Agora entendemos que é por isso mesmo que ele é executado usando esse artifício.

E agora você deve estar se perguntando se é só para isso que esse comando serve. Eu lhe digo que sim, mas isso nos traz um monte de vantagens – e uma das mais usadas é tratar funções como rotinas externas. Veja na **listagem 5** uma outra forma de fazer o nosso programa para incluir CDs no arquivo musicas.

Agora o programa deu uma boa encolhida e as chamadas de função foram trocadas por arquivos externos chamados `pergunta.func` e `mandamsg.func`, que assim podem ser chamados por qualquer outro programa, dessa forma reutilizando o seu código.

Por motivos meramente didáticos, as chamadas a `pergunta.func` e `mandamsg.func` estão sendo feitas por *source* e por `.` (ponto) indiscriminadamente, embora eu prefira o *source* que, por ser mais visível, melhora a legibilidade do código e facilita sua posterior manutenção. Veja na listagem 6 como ficaram esses dois arquivos.

Em ambos os arquivos, fiz somente duas mudanças, que veremos nas observações a seguir. Porém, tenho mais três observações a fazer:

1. As variáveis não estão sendo mais declaradas como locais, porque essa é uma diretiva que só pode ser usada no corpo de funções e, portanto, essas variáveis permanecem no ambiente do shell, poluindo-o;
2. O comando *return* não está mais presente, mas poderia estar sem alterar em nada a lógica do script, uma vez que só serviria para indicar um eventual erro por meio de um código de retorno previamente estabelecido (por exemplo `return 1`, `return 2`, ...), sendo que o `return` e `return 0` são idênticos e significam que a rotina foi executada sem erros;

3. O comando que estamos acostumados a usar para gerar um código de retorno é o *exit*, mas a saída de uma rotina externa não pode ser feita dessa forma porque, como ela está sendo executada no mesmo shell do script que o chamou, o *exit* simplesmente encerraria esse shell, terminando a execução de todo o script;
4. De onde veio a variável `LinhaMesg`? Ela veio do script `musinc7.sh`, porque havia sido declarada antes da chamada das rotinas (nunca esqueça que o shell que está interpretando o script e essas rotinas é o mesmo);
5. Se você decidir usar rotinas externas não se envergonhe, exagere nos comentários, principalmente sobre a passagem dos parâmetros, para facilitar a manutenção e o uso dessa rotina por outros programas no futuro.

– Bem, agora você já tem mais um monte de novidades para melhorar os scripts que fizemos. Você se lembra do programa *listartista.sh* no qual você passava o nome de um artista como parâmetro e ele devolvia as suas músicas? Ele era como o mostrado aqui embaixo na **listagem 7**.

– Claro que me lembro!

– Para firmar os conceitos que te passei, faça-o com a tela formatada e a execução em *loop*, de forma que ele só termine quando receber um ***Enter*** no lugar do nome do artista. Suponha que a tela tenha 25 linhas; a cada 22 músicas listadas o programa deverá dar uma parada para que o operador possa lê-las. Eventuais mensagens de erro devem ser passadas usando a rotina `mandamsg.func` que acabamos de desenvolver. Chico, manda mais dois!! O meu é com pouca pressão...

Listagem 6: *pergunta.func* e *mandamsg.func*

```
01 $ cat pergunta.func
02 #  A função recebe 3 parâmetros na seguinte ordem:
03 #  $1 - Mensagem a ser dada na tela
04 #  $2 - Valor a ser aceito com resposta default
05 #  $3 - O outro valor aceito
06 #  Supondo que $1=Aceita?, $2=s e $3=n, a linha
07 #  abaixo colocaria em Msg o valor "Aceita? (S/n)"
08 Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
09 TamMsg=${#Msg}
10 Col=$(((TotCols - TamMsg) / 2))    # Centraliza msg na linha
11 tput cup $LinhaMesg $Col
12 echo "$Msg"
13 tput cup $LinhaMesg $((Col + TamMsg + 1))
14 read -n1 SN
15 [ ! $SN ] && SN=$2                # Se vazia coloca default em SN
16 echo $SN | tr A-Z a-z             # A saída de SN será em minúscula
17 tput cup $LinhaMesg $Col; tput el  # Apaga msg da tela
18 $ cat mandamsg.func
19 # A função recebe somente um parâmetro
20 # com a mensagem que se deseja exibir;
21 # para não obrigar o programador a passar
22 # a msg entre aspas, usaremos $* (todos
23 # os parâmetro, lembra?) e não $1.
24 Msg="$*"
25 TamMsg=${#Msg}
26 Col=$(((TotCols - TamMsg) / 2))    # Centraliza msg na linha
27 tput cup $LinhaMesg $Col
28 echo "$Msg"
29 read -n1
30 tput cup $LinhaMesg $Col; tput el  # Apaga msg da tela
```

Não se esqueça: em caso de dúvida ou falta de companhia para um chope, é só mandar um e-mail para o endereço julio.neves@gmail.com que terei prazer em lhe ajudar. Vou aproveitar também para mandar minha propaganda: diga aos amigos que quem estiver a fim de fazer um curso porreta de programação em Shell deve mandar um e-mail para julio.neves@tecnohall.com.br para informar-se. Até mais! ■

Listagem 7: *listartista.sh*

```
01 $ cat listartista.sh
02 #!/bin/bash
03 # Dado um artista, mostra as suas musicas
04 # versao 2
05 if  [ $# -eq 0 ]
06 then
07     echo Voce deveria ter passado pelo menos um parametro
08     exit 1
09 fi
10 IFS="
11 :"
12 for ArtMus in $(cut -f2 -d^ musicas)
13 do
14     echo "$ArtMus" | grep -i "^$*~" > /dev/null && echo $ArtMus | cut -f2 -d~
15 done
```

Informações

[1] *Bash*, página oficial: http://www.gnu.org/software/bash/bash.html

[2] Manual de referência do *Bash*: http://www.gnu.org/software/bash/manual/bashref.html

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

Curso de Shell Script

Papo de Botequim
Parte IX

Hoje vamos aprender mais sobre formatação de cadeias de caracteres, conhecer as principais variáveis do Shell e nos aventurar no (ainda) desconhecido território da expansão de parâmetros. E dá-lhe chope!

por Júlio Cezar Neves

Tá bom, já sei que você vai querer chope antes de começar, mas tô tão a fim de te mostrar o que fiz que já vou pedir a rodada enquanto isso. Aê Chico, manda dois! O dele é sem colarinho pra não deixar cheiro ruim nesse bigodão...

Enquanto o chope não chega, deixa eu te lembrar o que você me pediu na edição passada: era para refazer o programa *listartista* com a tela formatada e execução em loop, de forma que ele só termine quando receber um ***[ENTER]*** sozinho como nome do artista. Eventuais mensagens de erro e perguntas feitas ao usuário deveriam ser mostradas na antepenúltima linha da tela, utilizando para isso as rotinas externas `mandamsg.func` e `pergunta.func` que desenvolvemos durante nosso papo na edição passada.

Primeiramente eu dei uma encolhida nas rotinas `mandamsg.func` e `pergunta.func`, que ficaram como na **listagem 1**. E na **listagem 2** você tem o "grandão", nossa versão refeita do *listaartista*.

– Puxa, você chegou com a corda toda! Gostei da forma como você resolveu o problema e estruturou o programa. Foi mais trabalhoso, mas a apresentação ficou muito legal e você explorou bastante as opções do comando *tput*. Vamos testar o resultado com um álbum do *Emerson, Lake & Palmer* que tenho cadastrado.

Envenenando a escrita

Ufa! Agora você já sabe tudo sobre leitura de dados, mas quanto à escrita ainda está apenas engatinhando. Já sei que você vai me perguntar: "Ora, não é com o comando *echo* e com os redirecionamentos de saída que se escreve dados?".

Bom, a resposta é "mais ou menos". Com estes comandos você escreve 90% do que precisa, porém se precisar escrever algo formatado eles lhe darão muito trabalho. Para formatar a saída veremos agora uma instrução muito mais interessante, a *printf*. Sua sintaxe é a seguinte:

Listagem 1: mandamsg.func e pergunta.func

mandamsg.func

```
01 # A  função recebe  somente um parâmetro
02 # com a  mensagem que se deseja exibir.
03 # Para não obrigar o programador a passar
04 # a msg entre aspas, usaremos $* (todos
05 # os parâmetro, lembra?) e não $1.
06 Msg="$*"
07 TamMsg=${#Msg}
08 Col=$(((TotCols - TamMsg) / 2))    # Centra msg na linha
09 tput cup $LinhaMesg $Col
10 read -n1 -p "$Msg "
```

pergunta.func

```
01 #  A função recebe 3 parâmetros na seguinte ordem:
02 #  $1 - Mensagem a ser mostrada na tela
03 #  $2 - Valor a ser aceito com resposta padrão
04 #  $3 - O outro valor aceito
05 #  Supondo que $1=Aceita?, $2=s e $3=n, a linha
06 #  abaixo colocaria em Msg o valor "Aceita? (S/n)"
07 Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
08 TamMsg=${#Msg}
09 Col=$(((TotCols - TamMsg) / 2))    # Centraliza msg na linha
10 tput cup $LinhaMesg $Col
11 read -n1 -p "$Msg " SN
12 [ ! $SN ] && SN=$2                 # Se vazia coloca default em SN
13 SN=$(echo $SN | tr A-Z a-z)        # A saída de SN será em minúscula
14 tput cup $LinhaMesg $Col; tput el  # Apaga msg da tela
```

```
printf formato [argumento...]
```

Onde *formato* é uma cadeia de caracteres que contém três tipos de objeto: caracteres simples, caracteres para especificação de formato (ou *de controle*) e seqüência de escape no padrão da linguagem C. *argumento* é a cadeia de caracteres a ser impressa sob o controle de *formato*.

Cada um dos caracteres utilizados é precedido pelo caracter % e, logo a seguir, vem a especificação de formato de acordo com a **tabela 1**.

As seqüências de escape padrão da linguagem C são sempre precedidas pelo caractere contra-barra (\). As reconhecidas pelo comando printf são as da **tabela 2**.

Não acaba por aí não! Tem muito mais coisa sobre essa instrução, mas como esse é um assunto muito cheio de detalhes e, portanto, chato para explicar e pior ainda para ler ou estudar, vamos passar direto aos exemplos com comentários. Veja só:

```
$ printf "%c" "1 caracter"
1$
```

Errado! Só listou 1 caractere e não saltou linha ao final

```
$ printf "%c\n" "1 caracter"
1
```

Saltou linha mas ainda não listou a cadeia inteira

```
$ printf "%c caractere\n" 1
1 caractere
```

Listagem 2: listaartista

```
$ cat listartista3.sh
01 #!/bin/bash
02 # Dado um artista, mostra as suas musicas
03 # versao 3
04 LinhaMesg=$((`tput lines` - 3))  # Linha onde as msgs serão mostradas
05 TotCols=$(tput cols)             # Qtd de colunas na tela para enquadrar msgs
06 clear
07 echo "
                          +--------------------------------------------------+
                          |  Lista Todas as Músicas de um Determinado Artista  |
                          |  ===== ===== == ======= == == ========== =======  |
                          |                                                  |
                          |  Informe o Artista:                              |
                          +--------------------------------------------------+"
08 while true
09 do
10     tput cup 5 51; tput ech 31  # ech=Erase chars (31 para não apagar barra vertical)
11     read Nome
12     if  [ ! "$Nome" ]           # $Nome esta vazio?
13     then
14         . pergunta.func "Deseja Sair?" s n
15         [ $SN = n ] && continue
16         break
17     fi
18     fgrep -iq "^$Nome~" musicas || # fgrep não interpreta ^ como expressão regular
19         {
20         . mandamsg.func "Não existe música desse artista"
21         continue
22         }
23     tput cup 7 29; echo '|                                                |'
24     LinAtual=8
25     IFS="
26 :"
27     for ArtMus in $(cut -f2 -d^ musicas) # Exclui nome do album
28     do
29         if  echo "$ArtMus" | grep -iq "^$Nome~"
30         then
31             tput cup $LinAtual 29
32             echo -n '|  "
33             echo $ArtMus | cut -f2 -d~
34             tput cup $LinAtual 82
35             echo '|'
36             let LinAtual++
37             if  [ $LinAtual -eq $LinhaMesg ]
38             then
39                 . mandamsg.func "Tecle Algo para Continuar..."
40                 tput cup 7 0; tput ed  # Apaga a tela a partir da linha 7
41                 tput cup 7 29; echo '|                                          |'
42                 LinAtual=8
43             fi
44         fi
45     done
46     tput cup $LinAtual 29; echo '|                                            |'
47     tput cup $((++LinAtual)) 29
48     read -n1 -p "+-----Tecle Algo para Nova Consulta----+"
49     tput cup 7 0; tput ed              # Apaga a tela a partir da linha 7
50 done
```

Tabela 1: Formatos de caractere

Caractere	A expressão será impressa como:
c	Caractere simples
d	Número no sistema decimal
e	Notação científica exponencial
f	Número com ponto decimal (*float*)
g	O menor entre os formatos %e e %f com omissão dos zeros não significativos
o	Número no sistema octal
s	Cadeia de caracteres
x	Número no sistema hexadecimal
%	Imprime um %. Não há nenhum tipo de conversão

Opa, essa é a forma correta! O `%c` recebeu o valor `1`, como queríamos:

```
$ a=2
$ printf "%c caracteres\n" $a
2 caracteres
```

O `%c` recebeu o valor da variável `$a`.

```
$ printf "%10c caracteres\n" $a
         2 caracteres
$ printf "%10c\n" $a caracteres
         2
         c
```

Repare que, nos dois últimos exemplos, em virtude do uso do `%c`, só foi listado um caractere de cada cadeia de caracteres passada como parâmetro. O valor `10` à frente do `c` não significa 10 caracteres. Um número seguindo o sinal de percentagem (`%`) significa o tamanho que a cadeia terá após a execução do comando. Vamos ver a seguir mais alguns exemplos. Os comandos abaixo:

Tabela 2: Seqüências de escape

Seqüência	Efeito
a	Soa o beep
b	Volta uma posição (*backspace*)
f	Salta para a próxima página lógica (*form feed*)
n	Salta para o início da linha seguinte (*line feed*)
r	Volta para o início da linha corrente (*carriage return*)
t	Avança para a próxima marca de tabulação

```
$ printf "%d\n" 32
32
$ printf "%10d\n" 32
        32
```

preenchem a *string* com espaços em branco à esquerda (oito espaços mais dois caracteres, 10 dígitos), não com zeros. Já no comando abaixo:

```
$ printf "%04d\n" 32
0032
```

O `04` após `%` significa "formate a string em quatro dígitos, com zeros à esquerda se necessário". No comando:

```
$ printf "%e\n" $(echo "scale=2 ; 100/6" | bc)
1.666000e+01
```

O padrão do `%e` é seis casas decimais. Já no comando:

```
$ printf "%.2e\n" `echo "scale=2 ; 100/6" | bc`
1.67e+01
```

O `.2` especificou duas casas decimais. Observe agora:

```
$ printf "%f\n" 32.3
32.300000
```

O padrão do `%f` é seis casas decimais. E no comando:

```
$ printf "%.2f\n" 32.3
32.30
```

O `.2` especificou duas casas decimais. Agora observe:

```
$ printf "%.3f\n" `echo "scale=2 ; 100/6" | bc`
33.330
```

O `bc` devolveu duas casas decimais e o `printf` colocou o zero à direita. O comando a seguir:

```
$ printf "%o\n" 10
12
```

Converteu o valor `10` para base octal. Para melhorar experimente:

```
$ printf "%03o\n" 27
033
```

Assim a conversão fica com mais jeito de octal, né?. O que este aqui faz?

```
$ printf "%s\n" Peteleca
Peteleca
$ printf "%15s\n" Peteleca
       Peteleca
```

Imprime `Peteleca` com 15 caracteres. A cadeia de caracteres é preenchida com espaços em branco à esquerda. Já no comando:

```
$ printf "%-15sNeves\n" Peteleca
Peteleca       Neves
```

O menos (`-`) colocou espaços em branco à direita de `Peteleca` até completar os 15 caracteres pedidos. E o comando abaixo, o que faz?

```
$ printf "%.3s\n" Peteleca
Pet
```

O `.3` manda truncar a cadeia de caracteres após as três primeiras letras. E o comando a seguir:

```
$ printf "%10.3sa\n" Peteleca
       Peta Pet
```

Imprime a cadeia com 10 caracteres, truncada após os três primeiros, concatenada com o caractere `a` (após o `s`). E esse comando a seguir, o que faz?

```
$ printf "EXEMPLO %x\n" 45232
EXEMPLO b0b0
```

Ele transformou o número 45232 para hexadecimal (b0b0), mas os zeros não combinam com o resto. Experimente:

```
$ printf "EXEMPLO %X\n" 45232
EXEMPLO B0B0
```

Assim disfarçou melhor! (repare no X maiúsculo). Pra terminar, que tal o comando abaixo:

```
$ printf "%X %XL%X\n" 49354 192 10
C0CA COLA
```

Este aí não é marketing e é bastante completo, veja só como funciona:

O primeiro %X converteu 49354 em hexadecimal, resultando em C0CA (leia-se "cê", "zero", "cê" e "a"). Em seguida veio um espaço em branco seguido por outro %XL. O %X converteu o 192 dando como resultado C0 que com o L fez COL. E finalmente o último parâmetro %X transformou o número 10 na letra A.

Conforme vocês podem notar, a instrução é bastante completa e complexa. Ainda bem que o echo resolve quase tudo...

Acertei em cheio quando resolvi explicar o printf através de exemplos, pois não saberia como enumerar tantas regrinhas sem tornar a leitura enfadonha.

Principais variáveis do Shell

O Bash possui diversas variáveis que servem para dar informações sobre o ambiente ou alterá-lo. São muitas e não pretendo mostrar todas elas, mas uma pequena parte pode lhe ajudar na elaboração de scripts. Veja a seguir as principais delas:

CDPATH » Contém os caminhos que serão pesquisados para tentar localizar um diretório especificado. Apesar dessa variável ser pouco conhecida, seu uso deve ser incentivado por poupar muito trabalho, principalmente em instalações com estrutura de diretórios em múltiplos níveis. Veja o exemplo a seguir:

```
$ echo $CDPATH
.:..:~:/usr/local
$ pwd
/home/jneves/LM
$ cd bin
$ pwd
/usr/local/bin
```

Como /usr/local estava na minha variável $CDPATH e não existia o diretório bin em nenhum dos seus antecessores (., .. e ~), o comando cd foi executado tendo como destino /usr/local/bin.

HISTSIZE » Limita o número de instruções que cabem dentro do arquivo de histórico de comandos (normalmente .bash_history, mas na verdade é o que está indicado na variável $HISTFILE). Seu valor padrão é 500.

HOSTNAME » O nome do host corrente (que também pode ser obtido com o comando uname -n).

LANG » Usada para determinar o idioma falado no país (mais especificamente categoria do *locale*). Veja um exemplo:

```
$ date
Thu Apr 14 11:54:13 BRT 2005
$ LANG=pt_BR date
Qui Abr 14 11:55:14 BRT 2005
```

LINENO » O número da linha do script ou função que está sendo executada. Seu uso principal é mostrar mensagens de erro juntamente com as variáveis $0 (nome do programa) e $FUNCNAME (nome da função em execução).

LOGNAME » Esta variável armazena o nome de login do usuário .

MAILCHECK » Especifica, em segundos, a freqüência com que o Shell verifica a presença de correspondência nos arquivos indicados pela variáveis $MAILPATH ou $MAIL. O tempo padrão é de 60 segundos (1 minuto). A cada intervalo o Shell fará a verificação antes de exibir o próximo prompt primário ($PS1). Se essa variável estiver sem valor ou com um valor menor ou igual a zero, a busca por novas mensagens não será efetuada.

PATH » Caminhos que serão pesquisados para tentar localizar um arquivo especificado. Como cada script é um arquivo, caso use o diretório corrente (.) na sua variável $PATH, você não necessitará usar o comando ./scrp para que o script scrp seja executado. Basta digitar scrp. Este é o modo que prefiro.

PIPESTATUS » É uma variável do tipo vetor (*array*) que contém uma lista de valores de códigos de retorno do último pipeline executado, isto é, um array que abriga cada um dos $? de cada instrução do último pipeline. Para entender melhor, veja o exemplo a seguir:

```
$ who
jneves   pts/0   Apr 11 16:26 (10.2.4.144)
jneves   pts/1   Apr 12 12:04 (10.2.4.144)
$ who | grep ^botelho
$ echo ${PIPESTATUS[*]}
0 1
```

Neste exemplo mostramos que o usuário botelho não estava "logado", em seguida executamos um pipeline que procurava por ele. Usa-se a notação [*] em um array para listar todos os seus elementos; dessa forma, vimos que a primeira instrução (who) foi bem-sucedida (código de retorno 0) e a seguinte (grep) não (código de retorno 1).

PROMPT_COMMAND » Se esta variável receber o nome de um comando, toda vez que você teclar um ***[ENTER]*** sozinho no prompt principal ($PS1), esse comando será executado. É muito útil quando você precisa repetindo constantemente uma determinada instrução.

PS1 » É o prompt principal. No "Papo de Botequim" usamos os padrões $ para usuário comum e # para root, mas é mui-

to freqüente que ele esteja personalizado. Uma curiosidade é que existe até concurso de quem programa o `$PS1` mais criativo.

`PS2` » Também chamado "prompt de continuação", é aquele sinal de maior (>) que aparece após um ***[ENTER]*** sem o comando ter sido encerrado.

`PWD` » Possui o caminho completo (`$PATH`) do diretório corrente. Tem o mesmo efeito do comando `pwd`.

`RANDOM` » Cada vez que esta variável é acessada, devolve um inteiro aleatório entre 0 e 32767. Para gerar um inteiro entre 0 e 100, por exemplo, digitamos:

```
$ echo $((RANDOM%101))
73
```

Ou seja, pegamos o resto da divisão do número randômico gerado por 101 porque o resto da divisão de qualquer número por 101 varia entre 0 e 100.

`REPLY` » Use esta variável para recuperar o último campo lido, caso ele não tenha nenhuma variável associada. Exemplo:

```
$ read -p "Digite S ou N: "
Digite S ou N: N
$ echo $REPLY
N
```

`SECONDS` » Esta variável informa, em segundos, há quanto tempo o Shell corrente está "de pé". Use-a para demonstrar a estabilidade do Linux e esnobar usuários daquela coisa com janelinhas coloridas que chamam de sistema operacional, mas que necessita de "reboots" freqüentes.

`TMOUT` » Se esta variável contiver um valor maior do que zero, esse valor será considerado o *timeout* padrão do comando *read*. No prompt, esse valor é interpretado como o tempo de espera por uma ação antes de expirar a sessão. Supondo que a variável contenha o valor `30`, o Shell encerrará a sessão do usuário (ou seja, fará *logout*) após 30 segundos sem nenhuma ação no prompt.

Expansão de parâmetros

Bem, muito do que vimos até agora são comandos externos ao Shell. Eles quebram o maior galho, facilitam a visualização, manutenção e depuração do código, mas não são tão eficientes quanto os intrínsecos (*built-ins*). Quando o nosso problema for performance, devemos dar preferência ao uso dos intrínsecos. A partir de agora vou te mostrar algumas técnicas para o seu programa pisar no acelerador.

Na **tabela 3** e nos exemplos a seguir, veremos uma série de construções chamadas expansão (ou substituição) de parâmetros (*Parameter Expansion*), que substituem instruções como o `cut`, o `expr`, o `tr`, o `sed` e outras de forma mais ágil.

Vamos ver alguns exemplos: se em uma pergunta o `S` é oferecido como valor *default* (padrão) e a saída vai para a variável `SN`, após ler o valor podemos fazer:

```
SN=${SN:-S}
```

Para saber o tamanho de uma `cadeia`:

```
$ cadeia=0123
$ echo ${#cadeia}
4
```

Para extrair dados de `cadeia`, da posição um até o final fazemos:

```
$ cadeia=abcdef
$ echo ${cadeia:1}
bcdef
```

Repare que a origem é zero e não um. Vamos extrair 3 caracteres a partir da 2ª posição da mesma variável `cadeia`:

```
$ echo ${cadeia:2:3}
cde
```

Repare que novamente a origem da contagem é zero e não um. Para suprimir tudo à esquerda da primeira ocorrência de uma cadeia, faça:

```
$ cadeia="Papo de Botequim"
$ echo ${cadeia#*' '}
de Botequim
$ echo "Conversa "${cadeia#*' '}
Conversa de Botequim
```

No exemplo anterior foi suprimido à esquerda tudo o que "casa" com a menor ocorrência da expressão `*' '`, ou seja, todos os caracteres até o primeiro espaço em branco. Esses exemplos também poderiam ser escritos sem proteger o espaço da interpretação do Shell (mas prefiro protegê-lo para facilitar a legibilidade do código). Veja só:

```
$ echo ${cadeia#* }
de Botequim
$ echo "Conversa "${cadeia#* }
Conversa de Botequim
```

Repare que na construção de `expr` é permitido o uso de metacaracteres.

Utilizando o mesmo valor da variável `cadeia`, observe como faríamos para ter somente `Botequim`:

```
$ echo ${cadeia##*' '}
Botequim
$ echo "Vamos 'Chopear' no "${cadeia##*' '}
Vamos 'Chopear' no Botequim
```

Desta vez suprimimos à esquerda de `cadeia` a maior ocorrência da expressão `expr`. Assim como no caso anterior, o uso de metacaracteres é permitido.

Outro exemplo mais útil: para que não apareça o caminho (*path*) completo do seu programa (`$0`) em uma mensagem de erro, inicie o seu texto da seguinte forma:

```
echo Uso: ${0##*/} texto da mensagem de erro
```

Neste exemplo seria suprimido à esquerda tudo até a última barra (`/`) do caminho, restando somente o nome do programa. O caractere `%` é simétrico ao `#`, veja o exemplo:

Tabela 3: Tipos de expansão de parâmetros

Expansão de parâmetros	Resultado esperado
${var:-padrao}	Se `var` é vazia, o resultado da expressão é padrão
${#cadeia}	Tamanho de `$cadeia`
${cadeia:posicao}	Extrai uma subcadeia de `$cadeia` a partir de posição. Origem zero
${cadeia:posicao:tamanho}	Extrai uma subcadeia de `$cadeia` a partir de posição com tamanho igual a `tamanho`. Origem zero
${cadeia#expr}	Corta a menor ocorrência de `$cadeia` à esquerda da expressão `expr`
${cadeia##expr}	Corta a maior ocorrência de `$cadeia` à esquerda da expressão `expr`
${cadeia%expr}	Corta a menor ocorrência de `$cadeia` à direita da expressão `expr`
${cadeia%%expr}	Corta a maior ocorrência de `$cadeia` à direita da expressão `expr`
${cadeia/subcad1/subcad2}	Troca a primeira ocorrência de `subcad1` por `subcad2`
${cadeia//subcad1/subcad2}	Troca todas as ocorrências de `subcad1` por `subcad2`
${cadeia/#subcad1/subcad2}	Se `subcad1` combina com o início de cadeia, então é trocado por `subcad2`
${cadeia/%subcad1/subcad2}	Se `subcad1` combina com o fim de cadeia, então é trocado por `subcad2`

```
$ echo $cadeia
Papo de Botequim
$ echo ${cadeia%' '*}
Papo de
$ echo ${cadeia%%' '*}
Papo
```

Para trocar primeira ocorrência de uma subcadeia em uma cadeia por outra:

```
$ echo $cadeia
Papo de Botequim
$ echo ${cadeia/de/no}
Papo no Botequim
$ echo ${cadeia/de /}
Papo Botequim
```

Preste atenção quando for usar metacaracteres! Eles são gulosos e sempre combinarão com a maior possibilidade; No exemplo a seguir eu queria trocar *Papo de Botequim* por *Conversa de Botequim*:

```
$ echo $cadeia
Papo de Botequim
$ echo ${cadeia/*o/Conversa}
Conversatequim
```

A idéia era pegar tudo até o primeiro `o`, mas acabou sendo trocado tudo até o último `o`. Isto poderia ser resolvido de diversas maneiras. Eis algumas:

```
$ echo ${cadeia/*po/Conversa}
Conversa de Botequim
$ echo ${cadeia/????/Conversa}
Conversa de Botequim
```

Trocando todas as ocorrências de uma subcadeia por outra. O comando:

```
$ echo ${cadeia//o/a}
Papa de Batequim
```

Ordena a troca de todos as letras `o` por `a`. Outro exemplo mais útil é para contar a quantidade de arquivos existentes no diretório corrente. Observe o exemplo:

```
$ ls | wc -l
      30
```

O `wc` põe um monte de espaços em branco antes do resultado. Para tirá-los:

```
# QtdArqs recebe a saída do comando
$ QtdArqs=$(ls | wc -l)
$ echo ${QtdArqs/ * /}
30
```

Nesse exemplo, eu sabia que a saída era composta de brancos e números, por isso montei essa expressão para trocar todos os espaços por nada. Note que antes e após o asterisco (*) há espaços em branco.

Há várias formas de trocar uma subcadeia no início ou no fim de uma variável. Para trocar no início fazemos:

```
$ echo $Passaro
quero quero
$ echo "Como diz o sulista - "${PassaroU
/#quero/não}
Como diz o sulista - não quero
```

Para trocar no final fazemos:

```
$ echo "Como diz o nordestino - "
"${Passaro/%quero/não}
Como diz o nordestino - quero não
```

– Agora já chega, o papo hoje foi chato porque teve muita decoreba, mas o que mais importa é você ter entendido o que te falei. Quando precisar, consulte estes guardanapos onde rabisquei as dicas e depois guarde-os para consultas futuras. Mas voltando à vaca fria: tá na hora de tomar outro e ver o jogo do Mengão. Pra próxima vez vou te dar moleza e só vou cobrar o seguinte: pegue a rotina `pergunta.func` (da qual falamos no início do nosso bate-papo de hoje, veja a **listagem 1**) e otimize-a para que a variável `$SN` receba o valor padrão por expansão de parâmetros, como vimos.

E não se esqueça: em caso de dúvidas ou falta de companhia para um (ou mais) chope é só mandar um e-mail para julio.neves@gmail.com. E diga para os amigos que quem estiver a fim de fazer um curso porreta de programação em Shell deve mandar um e-mail para julio.neves@tecnohall.com.br para informar-se. Valeu! ■

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

Curso de Shell Script

Papo de Botequim
Parte X

Em mais um capítulo de nossa saga através do mundo do *Shell Script*, vamos aprender a avaliar expressões, capturar sinais e receber parâmetros através da linha de comando.

por Júlio Cezar Neves

Eaê amigo, te dei a maior moleza na última aula né? Um exerciciozinho muito simples...

– É, mas nos testes que eu fiz, e de acordo com o que você ensinou sobre substituição de parâmetros, achei que deveria fazer algumas alterações nas funções que desenvolvemos para torná-las de uso geral, como você disse que todas as funções deveriam ser. Quer ver?

– Claro, né, mané, se te pedi para fazer é porque estou a fim de te ver aprender, mas peraí, dá um tempo. Chico! Manda dois, um sem colarinho! Vai, mostra aí o que você fez.

– Bem, além do que você pediu, eu reparei que o programa que chamava a função teria de ter previamente definidas a linha em que seria mostrada a mensagem e a quantidade de colunas. O que fiz foi incluir duas linhas – nas quais empreguei substituição de parâmetros – para que, caso uma dessas variáveis não fosse informada, ela recebesse um valor atribuído pela própria função. A linha de mensagem é três linhas antes do fim da tela e o total de colunas é obtido pelo comando `tput cols`. Dê uma olhada na **listagem 1** e veja como ficou:

– Gostei, você já se antecipou ao que eu ia pedir. Só pra gente encerrar esse papo de substituição de parâmetros, repare que a legibilidade do código está "horrorível", mas o desempenho, isto é, a velocidade de execução, está ótimo. Como funções são coisas muito pessoais, já que cada um usa as suas e quase não há necessidade de manutenção, eu sempre opto pelo desempenho.

Hoje vamos sair daquela chatura que foi o nosso último papo e voltar à lógica, saindo da decoreba. Mas volto a te lembrar: tudo que eu te mostrei da última vez aqui no Boteco do Chico é válido e quebra um galhão. Guarde aqueles guardanapos que rabiscamos porque, mais cedo ou mais tarde, eles lhe vão ser muito úteis.

O comando eval

Vou te dar um problema que eu duvido que você resolva:

```
$ var1=3
$ var2=var1
```

Te dei essas duas variáveis e quero que você me diga como eu posso, me referindo apenas à variável a `var2`, listar o valor de `var1` (que, no nosso caso, é 3).

– Ah, isso é mole, mole! É só digitar esse comando aqui:

```
echo $`echo $var2`
```

Listagem 1: função pergunta.func

```
01 #  A função recebe 3 parâmetros na seguinte ordem:
02 #  $1 - Mensagem a ser mostrada na tela
03 #  $2 - Valor a ser aceito com resposta padrão
04 #  $3 - O outro valor aceito
05 #  Supondo que $1=Aceita?, $2=s e $3=n, a linha
06 #  abaixo colocaria em Msg o valor "Aceita? (S/n)"
07 TotCols=${TotCols:-$(tput cols)}   # Se não estava definido, agora está
08 LinhaMesg=${LinhaMesg:-$(($(tput lines)-3))} # Idem
09 Msg="$1 (`echo $2 | tr a-z A-Z`/`echo $3 | tr A-Z a-z`)"
10 TamMsg=${#Msg}
11 Col=$(((TotCols - TamMsg) / 2))    # Para centralizar Msg na linha
12 tput cup $LinhaMesg $Col
13 read -n1 -p "$Msg " SN
14 SN=${SN:-$2}                       # Se vazia coloca o padrão em SN
15 SN=$(echo $SN | tr A-Z a-z)        # A saída de SN será em minúsculas
16 tput cup $LinhaMesg $Col; tput el  # Apaga Msg da tela
```

Repare que eu coloquei o `echo $var2` entre crases (`), porque dessa forma ele terá prioridade de execução e resultará em `var1`. E `echo $var1` produzirá 3...

– Ah, é? Então execute para ver se está correto.

```
$ echo $`echo $var2`
$var1
```

– Ué! Que foi que aconteceu? O meu raciocínio me parecia bastante lógico...

– O seu raciocínio realmente foi lógico, o problema é que você esqueceu de uma das primeiras coisas de que te falei aqui no Boteco e que vou repetir. O Shell usa a seguinte ordem para resolver uma linha de comando:

- Resolve os redirecionamentos;
- Substitui as variáveis pelos seus valores;
- Resolve e substitui os meta caracteres;
- Passa a linha já toda esmiuçada para execução.

Dessa forma, quando o interpretador chegou na fase de resolução de variáveis, que como eu disse é anterior à execução, a única variável existente era `var2` e por isso a tua solução produziu como saída `$var1`. O comando *echo* identificou isso como uma cadeia de caracteres e não como uma variável.

Problemas desse tipo são relativamente freqüentes e seriam insolúveis caso não existisse a instrução *eval*, cuja sintaxe é `eval cmd`, onde `cmd` é uma linha de comando qualquer, que você poderia inclusive executar direto no prompt do terminal. Quando você põe o eval na frente, no entanto, o que ocorre é que o Shell trata `cmd` como um parâmetro do eval e, em seguida, o eval executa a linha recebida, submetendo-a ao Shell. Ou seja, na prática `cmd` é analisado duas vezes. Dessa forma, se executássemos o comando que você propôs colocando o eval na frente, teríamos a saída esperada. Veja:

```
$ eval echo $`echo $var2`
3
```

Esse exemplo também poderia ter sido feito de outra maneira. Dá só uma olhada:

```
$ eval echo \$$var2
3
```

Na primeira passada a contrabarra (\) seria retirada e `$var2` seria resolvido produzindo `var1`. Na segunda passada teria sobrado `echo $var1`, que produziria o resultado esperado. Agora vou colocar um comando dentro de `var2` e executar:

```
$ var2=ls
$ $var2
10porpag1.sh  alo2.sh        incusu        logado
10porpag2.sh  ArqDoDOS.txt1  listamusica   logaute.sh
10porpag3.sh  confuso        listartista   mandamsg.func
alo1.sh       contpal.sh     listartista3  monbg.sh
```

Agora vamos colocar em `var2` o seguinte: `ls $var1`; e em `var1` vamos colocar `l*`, vejamos o resultado:

```
$ var2='ls $var1'
$ var1='l*'
$ $var2
ls: $var1: No such file or directory
$ eval $var2
listamusica  listartista  listartista3  logado  logaute.sh
```

Novamente, no tempo de substituição das variáveis, `$var1` ainda não havia se apresentado ao Shell para ser resolvida. Assim, só nos resta executar o comando eval para dar as duas passadas necessárias.

Uma vez um colega da excelente lista de discussão groups.yahoo.com/group/shell-script colocou uma dúvida: queria fazer um menu que numerasse e listasse todos os arquivos com extensão `.sh` e, quando o operador escolhesse uma opção, o programa correspondente fosse executado. Veja minha proposta na **listagem 2**:

Listagem 2: fazmenu.sh

```
01 #!/bin/bash
02 #
03 # Lista que enumera os programas com extensão .sh no
04 # diretório corrente e executa o escolhido pelo operador
05 #
06 clear; i=1
07 printf "%11s\t%s\n\n" Opção Programa
08 CASE='case $opt in'
09 for arq in *.sh
10 do
11     printf "\t%03d\t%s\n" $i $arq
12     CASE="$CASE
13     "$(printf "%03d)\t %s;;" $i $arq)
14     i=$((i+1))
15 done
16 CASE="$CASE
17       *)        . erro;;
18 esac"
19 read -n3 -p "Informe a opção desejada: " opt
20 echo
21 eval "$CASE"
```

Parece complicado porque usei muitos `printf` para formatação da tela, mas na verdade é bastante simples: o primeiro `printf` foi colocado para imprimir o cabeçalho e logo em seguida comecei a montar dinamicamente a variável `$CASE`, na qual ao final será feito um eval para execução do programa escolhido. Repare no entanto que dentro do loop do `for` existem dois `printf`: o primeiro serve para formatar a tela e o segundo para montar o `case` (se antes do comando `read` você colocar uma linha `echo "$CASE"`, verá que o comando `case` montado dentro da variável está todo indentado. Frescura, né?:). Na saída do `for`, foi adicionada uma linha à variável `$CASE` para, no caso de uma escolha inválida, ser executada uma função externa para exibir mensagens de erro. Vamos executar o script para ver a saída gerada:

```
$ fazmenu.sh
      Opcao     Programa

        001     10porpag1.sh
        002     10porpag2.sh
        003     10porpag3.sh
        004     alo1.sh
        005     alo2.sh
        006     contpal.sh
        007     fazmenu.sh
        008     logaute.sh
        009     monbg.sh
        010     readpipe.sh
        011     redirread.sh
Informe a opção desejada:
```

Seria interessante incluir uma opção para terminar o programa e, para isso, seria necessária a inclusão de uma linha após o loop de montagem da tela e a alteração da linha na qual fazemos a atribuição final do valor da variável `$CASE`. Veja na **listagem 3** como ele ficaria:

Existe no Linux uma coisa chamada sinal (`signal`). Existem diversos sinais que podem ser mandados para (ou gerados por) processos em execução. Vamos, de agora em diante, dar uma olhadinha nos sinais enviados aos processos e mais à frente vamos dar uma passada rápida pelos sinais gerados pelos processos. Para mandar um sinal a um processo, usamos normalmente o comando *kill*, cuja sintaxe é:

```
$ kill -sig PID
```

Onde `PID` é o identificador do processo (*Process Identification* ou *Process ID*). Além do comando kill, algumas seqüências de teclas também podem gerar sinais. A **tabela 1** mostra os sinais mais importantes para monitorarmos:

Além desses, existe o famigerado sinal `-9` ou `SIGKILL` que, para o processo que o está recebendo, equivale a meter o dedo no botão de desligar do computador – o que é altamente indesejável, já que muitos programas necessitam "limpar a área" ao seu término. Se seu encerramento ocorrer de forma prevista, ou seja, se tiver um término normal, é muito fácil fazer essa limpeza; porém, se o seu programa tiver um fim brusco, muita coisa ruim pode ocorrer:

- É possível que em um determinado espaço de tempo, o seu computador esteja cheio de arquivos de trabalho inúteis
- Seu processador poderá ficar atolado de processos *zombies* e *defuncts* gerados por processos filhos que perderam os pais e estão "órfãos";
- É necessário liberar *sockets* abertos para não deixar os clientes congelados;
- Seus bancos de dados poderão ficar corrompidos porque sistemas gerenciadores de bancos de dados necessitam de um tempo para gravar seus buffers em disco (commit).

Enfim, existem mil razões para não usar um kill com o sinal `-9` e para monitorar o encerramento anormal de programas.

Listagem 3: Nova versão do fazmenu.sh

```
01 #!/bin/bash
02 #
03 # Lista enumerando os programas com extensão .sh no
04 # diretório corrente; executa o escolhido pelo operador
05 #
06 clear; i=1
07 printf "%11s\t%s\n\n" Opção Programa
08 CASE='case $opt in'
09 for arq in *.sh
10 do
11     printf "\t%03d\t%s\n" $i $arq
12     CASE="$CASE
13     "$(printf "%03d)\t %s;;" $i $arq)
14     i=$((i+1))
15 done
16 printf "\t%d\t%s\n\n" 999 "Fim do programa"  # Linha incluída
17 CASE="$CASE
18     999)        exit;;                      # Linha alterada
19       *)        ./erro;;
20 esac"
21 read -n3 -p "Informe a opção desejada: " opt
22 echo
23 eval "$CASE"
```

O comando trap

Para fazer a monitoração de sinais existe o comando `trap`, cuja sintaxe pode ser uma das mostradas a seguir:

```
trap "cmd1; cmd2; cmdn" S1 S2 … SN
trap 'cmd1; cmd2; cmdn' S1 S2 … SN
```

Onde os comandos `cmd1, cmd2, cmdn` serão executados caso o programa receba os sinais `S1, S2 … SN`. As aspas (") ou as apóstrofes (') só são necessárias caso o *trap* possua mais de um comando `cmd` associado. Cada uma delas pode ser também uma função interna, uma externa ou outro script.

Para entender o uso de aspas (") e apóstrofes (') vamos recorrer a um exemplo que trata um fragmento de um script que faz uma transferência de arquivos via FTP para uma máquina remota (`$RemoComp`), na qual o usuário é `$Fulano`, sua senha é `$Segredo` e o arquivo a ser enviado é `$Arq`. Suponha ainda que essas quatro variáveis foram recebidas por uma rotina anterior de leitura e que esse script seja muito usado por diversas pessoas. Vejamos o trecho de código a seguir:

```
ftp -ivn $RemoComp << FimFTP >> /tmp/$$ 2
2>> /tmp/$$
    user $Fulano $Segredo
    binary
    get $Arq
FimFTP
```

Repare que tanto as saídas dos diálogos do FTP como os erros encontrados estão sendo redirecionados para `/tmp/$$`, o que é uma construção bastante comum para arquivos temporários usados em scripts com mais de um usuário, porque `$$` é a variável que contém o número do processo (PID), que é único. Com esse tipo de construção evita-se que dois ou mais usuários disputem a posse e os direitos sobre um arquivo.

Caso a transferência seja interrompida por um `kill` ou um ***[CTRL]+[C]***, certamente deixará lixo no disco. É exatamente essa a forma mais comum de uso do comando `trap`. Como isso é trecho de um script devemos, logo no início dele, digitar o comando:

```
trap "rm -f /tmp/$$ ; exit" 0 1 2 3 15
```

Dessa forma, caso houvesse uma interrupção brusca (sinais 1, 2 , 3 ou 15) antes do programa encerrar (no `exit` dentro do comando `trap`), ou um fim normal (sinal 0), o arquivo `/tmp/$$` seria removido.

Caso não houvesse a instrução `exit` na linha de comando do `trap`, ao final da execução dessa linha o fluxo do programa retornaria ao ponto em que estava quando recebeu o sinal que originou a execução desse `trap`.

Note também que o Shell pesquisa a linha de comando uma vez quando o `trap` é interpretado (e é por isso que é usual colocá-lo no início do programa) e novamente quando um dos sinais listados é recebido. Então, no último exemplo, o valor de `$$` será substituído no momento em que o comando `trap` é lido pela primeira vez, já que as aspas (") não protegem o cifrão (`$`) da interpretação do Shell.

Se você quisesse fazer a substituição somente ao receber o sinal, o comando deveria ser colocado entre apóstrofes ('). Assim, na primeira interpretação do `trap`, o Shell não veria o cifrão (`$`), as apóstrofes (') seriam removidas e, finalmente, o Shell poderia substituir o valor da variável. Nesse caso, a linha ficaria assim:

```
trap 'rm -f /tmp/$$ ; exit' 0 1 2 3 15
```

Suponha dois casos: você tem dois scripts que chamaremos de `script1`, cuja primeira linha será um `trap`, e `script2`, colocado em execução por `script1`. Por serem dois processos diferentes, terão dois PIDs distintos.

Tabela 1: Principais sinais

Código	Nome	Gerado por:
0	EXIT	Fim normal do programa
1	SIGHUP	Quando o programa recebe um `kill -HUP`
2	SIGINT	Interrupção pelo teclado. (***[CTRL]+[C]***)
3	SIGQUIT	Interrupção pelo teclado (***[CTRL]+[\]***)
15	SIGTERM	Quando o programa recebe um `kill` ou `kill -TERM`

1º Caso: O comando `ftp` encontra-se em `script1`. Nesse caso, o argumento do comando trap deveria vir entre aspas (") porque, caso ocorresse uma interrupção (***[CTRL]+[C]*** ou ***[CTRL]+[\]***) no `script2`, a linha só seria interpretada nesse momento e o PID do `script2` seria diferente do encontrado em `/tmp/$$` (não esqueça que `$$` é a variável que contém o PID do processo ativo);

2º Caso: O comando `ftp` encontra-se em `script2`. Nesse caso, o argumento do comando `trap` deveria estar entre apóstrofes ('), pois caso a interrupção se desse durante a execução de `script1`, o arquivo não teria sido criado; caso ela ocorresse durante a execução de `script2`, o valor de `$$` seria o PID desse processo, que coincidiria com o de `/tmp/$$`.

O comando `trap`, quando executado sem argumentos, lista os sinais que estão sendo monitorados no ambiente, bem como a linha de comando que será executada quando tais sinais forem recebidos. Se a linha de comandos do `trap` for nula (vazia), isso significa que os sinais especificados devem ser ignorados quando recebidos. Por exemplo, o comando `trap "" 2` especifica que o sinal de interrupção (***[CTRL]+[C]***) deve ser ignorado. No último exemplo, note que o primeiro argumento deve ser especificado para que o sinal seja ignorado e não é equivalente a escrever `trap 2`, cuja finalidade é retornar o sinal `2` ao seu estado padrão.

Se você ignorar um sinal, todos os sub-shells irão ignorá-lo. Portanto, se você especificar qual ação deve ser tomada quando receber um sinal, todos os sub-shells irão tomar a mesma ação quando receberem esse sinal. Ou seja, os sinais são automaticamente exportados. Para o sinal mostrado (sinal 2), isso significa que os sub-shells serão encerrados. Suponha que você execute o comando `trap "" 2` e então execute um sub-shell, que tornará a executar outro script como um sub-shell. Se for gerado um sinal de interrupção, este não terá efeito nem sobre o Shell principal nem sobre os sub-shell por ele chamados, já que todos eles ignorarão o sinal.

Em korn shell (*ksh*) não existe a opção `-s` do comando `read` para ler uma senha. O que costumamos fazer é usar usar o comando `stty` com a opção `-echo`, que inibe a escrita na tela até que se encontre um `stty echo` para restaurar essa escrita. Então, se estivéssemos usando o interpretador ksh, a leitura da senha teria que ser feita da seguinte forma:

```
echo -n "Senha: "
stty -echo
read Senha
stty echo
```

O problema com esse tipo de construção é que, caso o operador não soubesse a senha, ele provavelmente teclaria ***[CTRL]+[C]*** ou um ***[CTRL]+[\]*** durante a instrução `read` para descontinuar o programa e, caso agisse dessa forma, o seu terminal estaria sem *echo*. Para evitar que isso aconteça, o melhor a fazer é:

```
echo -n "Senha: "
trap "stty echo
      exit" 2 3
stty -echo
read Senha
stty echo
trap 2 3
```

Para terminar esse assunto, abra um console gráfico e escreva no prompt de comando o seguinte:

```
$ trap "echo Mudou o tamanho da janela" 28
```

Em seguida, pegue o mouse e arraste-o de forma a variar o tamanho da janela corrente. Surpreso? É o Shell orientado a eventos... Mais unzinho, porque não consigo resistir. Escreva isto:

```
$ trap "echo já era" 17
```

Em seguida digite:

```
$ sleep 3 &
```

Você acabou de criar um sub-shell que irá dormir durante três segundos em background. Ao fim desse tempo, você receberá a mensagem "já era", porque o sinal 17 é emitido a cada vez em que um sub-shell termina a sua execução. Para devolver esses sinais ao seu comportamento padrão, digite: `trap 17 28`.

Muito legal esse comando, né? Se você descobrir algum material bacana sobre uso de sinais, por favor me informe por email, porque é muito rara a literatura sobre o assunto.

Comando getopts

O comando *getopts* recupera as opções e seus argumentos de uma lista de parâmetros de acordo com a sintaxe POSIX.2, isto é, letras (ou números) após um sinal de menos (`-`) seguidas ou não de um argumento; no caso de somente letras (ou números), elas podem ser agrupadas. Você deve usar esse comando para "fatiar" opções e argumentos passados para o seu script.

A sintaxe é `getopts cadeiadeopcoes nome`. A `cadeiadeopcoes` deve explicitar uma cadeia de caracteres com todas as opções reconhecidas pelo script; assim, se ele reconhece as opções `-a` `-b` e `-c`, `cadeiadeopcoes` deve ser `abc`. Se você desejar que uma opção seja seguida por um argumento, ponha um sinal de dois pontos (`:`) depois da letra, como em `a:bc`. Isso diz ao getopts que a opção `-a` tem a forma `-a argumento`. Normalmente um ou mais espaços em branco separam o parâmetro da opção; no entanto, getopts também manipula parâmetros que vêm colados à opção como em `-aargumento`. `cadeiadeopcoes` não pode conter um sinal de interrogação (`?`).

O nome constante da linha de sintaxe acima define uma variável que receberá, a cada vez que o comando getopts for executado, o próximo dos parâmetros posicionais e o colocará na variável `nome`. getopts coloca uma interrogação (`?`) na variável definida em nome se achar uma opção não definida em `cadeiadeopcoes` ou se não achar o argumento esperado para uma determinada opção.

Como já sabemos, cada opção passada por uma linha de comandos tem um índice numérico; assim, a primeira opção estará contida em `$1`, a segunda em `$2` e assim por diante. Quando o getopts obtém uma opção, ele armazena o índice do próximo parâmetro a ser processado na variável `OPTIND`.

Quando uma opção tem um argumento associado (indicado pelo `:` na `cadeiadeopcoes`), getopts armazena o argumento na variável `OPTARG`. Se uma opção não possuir argumento ou se o argumento esperado não for encontrado, a variável `OPTARG` será "apagada" (com `unset`). O comando encerra sua execução quando:

- Encontra um parâmetro que não começa com um hífen (`-`).
- O parâmetro especial `--` indica o fim das opções.
- Quando encontra um erro (por exemplo, uma opção não reconhecida).

O exemplo da **listagem 4** é meramente didático, servindo para mostrar, em um pequeno fragmento de código, o uso pleno do comando.

Para entender melhor, vamos executar o script:

```
$ getoptst.sh -h -Pimpressora arq1 arq2
getopts fez a variavel OPT_LETRA igual a 'h'
        OPTARG eh ''
getopts fez a variavel OPT_LETRA igual a 'P'
        OPTARG eh 'impressora'
Dispensando os primeiros $OPTIND-1 = 2 argumentos
O que sobrou da linha de comandos foi 'arq1 arq2'
```

Dessa forma, sem ter muito trabalho, separei todas as opções com seus respectivos argumentos, deixando somente os parâmetros que foram passados pelo operador para posterior tratamento. Repare que, se tivéssemos escrito a linha de comando com o argumento (`impressora`) separado da opção (`-P`), o resultado seria exatamente o mesmo, exceto pelo `OPTIND`, já que nesse caso ele identifica um conjunto de três opções (ou argumentos) e, no anterior, somente dois. Veja só:

```
$ getoptst.sh -h -P impressora arq1 arq2
getopts fez a variavel OPT_LETRA igual a 'h'
        OPTARG eh ''
getopts fez a variavel OPT_LETRA igual a 'P'
        OPTARG eh 'impressora'
Dispensando os primeiros $OPTIND-1 = 3 argumentos
O que sobrou da linha de comandos foi 'arq1 arq2'
```

Repare, no exemplo a seguir, que se passarmos uma opção inválida a variável `$OPT_LETRA` receberá um ponto de interrogação (`?`) e a `$OPTARG` será "apagada" (`unset`).

```
$ getoptst.sh -f -Pimpressora arq1 arq2  #  A opção -f não é valida
./getoptst.sh: illegal option -- f
getopts fez a variavel OPT_LETRA igual a '?'
        OPTARG eh ''
getopts fez a variavel OPT_LETRA igual a 'P'
        OPTARG eh 'impressora'
Dispensando os primeiros $OPTIND-1 = 2 argumentos
O que sobrou da linha de comandos foi 'arq1 arq2'
```

– Me diz uma coisa: você não poderia ter usado um condicional com *case* para evitar o getopts?

– Poderia sim, mas para quê? Os comandos estão aí para serem usados... O exemplo foi didático, mas imagine um programa que aceitasse muitas opções e cujos parâmetros poderiam ou não estar colados às opções, sendo que as opções também poderiam ou não estar coladas: ia ser um case infernal! Com getopts, é só seguir os passos acima.

– É... Vendo dessa forma, acho que você tem razão. É porque eu já estou meio cansado com tanta informação nova na minha cabeça. Vamos tomar a saideira ou você ainda quer explicar alguma particularidade do Shell?

– Nem um nem outro, eu também já cansei mas hoje não vou tomar a saideira porque estou indo dar aula na UniRIO, que é a primeira universidade federal que está preparando seus alunos do curso de graduação em Informática para o uso de Software Livre. Mas antes vou te deixar um problema para te encucar: quando você varia o tamanho de uma janela do terminal, no centro dela não aparece dinamicamente, em vídeo reverso, a quantidade de linhas e colunas? Então! Eu quero que você reproduza isso usando a linguagem Shell. Chico, traz rapidinho a minha conta! Vou contar até um e se você não trouxer eu me mando!

Não se esqueça, qualquer dúvida ou falta de companhia para um chope é só mandar um email para julio.neves@gmail.com. Vou aproveitar também para mandar o meu jabá: diga para os amigos que quem estiver a fim de fazer um curso porreta de programação em Shell que mande um e-mail para julio.neves@tecnohall.com.br para informar-se. Valeu! ■

Listagem 4: getoptst.sh

```
01 $ cat getoptst.sh
02 #!/bin/sh
03
04 # Execute assim:
05 #
06 #       getoptst.sh -h -Pimpressora arq1 arq2
07 #
08 # e note que as informações de todas as opções são exibidas
09 #
10 # A cadeia 'P:h' diz que a opção -P é uma opção complexa
11 # e requer um argumento e que h é uma opção simples que não requer
12 # argumentos.
13
14 while getopts 'P:h' OPT_LETRA
15 do
16     echo "getopts fez a variavel OPT_LETRA igual a '$OPT_LETRA'"
17     echo "  OPTARG eh '$OPTARG'"
18 done
19 used_up=`expr $OPTIND - 1`
20 echo "Dispensando os primeiros \$OPTIND-1 = $used_up argumentos"
21 shift $used_up
22 echo "O que sobrou da linha de comandos foi '$*'"
```

Curso de Shell Script

Papo de botequim

Parte final

A conversa está boa, mas uma hora eles tem que sair do bar. Na última parte do nosso papo, falamos sobre pipes e sincronização entre processos.

por Júlio Cezar Neves

E aí rapaz, tudo bom?

– Beleza. Você se lembra de que da última vez você me pediu para fazer um programa que imprimisse dinamicamente, no centro da tela, a quantidade de linhas e colunas de um terminal sempre que o tamanho da janela variasse? Pois é, eu até que fiz, mas mesmo depois de quebrar muito a cabeça a aparência não ficou igual.

Listagem 1: tamtela.sh

```
01 #!/bin/bash
02 #
03 # Coloca no centro da tela, em video reverso,
04 # a quantidade de colunas e linhas
05 # quando o tamanho da tela eh alterado.
06 #
07 trap Muda 28   # 28 = sinal gerado pela mudanca no tamanho
08               # da tela e Muda eh a funcao que fara isso.
09
10 Bold=$(tput bold)    # Modo de enfase
11 Rev=$(tput rev)      # Modo de video reverso
12 Norm=$(tput sgr0)    # Restaura a tela ao padrao default
13
14 Muda ()
15 {
16    clear
17    Cols=$(tput cols)
18    Lins=$(tput lines)
19    tput cup $(($Lins / 2)) $(((Cols - 7) / 2)) # Centro da tela
20    echo $Bold$Rev$Cols X $Lins$Norm
21 }
22
23 clear
24 read -n1 -p "Mude o tamanho da tela ou tecle algo para terminar "
```

– Não estou nem aí para a aparência, o que eu queria é que você exercitasse o que aprendemos. Me dá a **listagem 1** pra eu ver o que você fez.

– Perfeito! Que se dane a aparência, depois vou te ensinar uns macetes para melhorá-la. O que vale é que o programa está funcionando e está bem enxuto.

– Pôxa, e eu que perdi o maior tempo tentando descobrir como aumentar a fonte...

– Deixe isso para lá! Hoje vamos ver umas coisas bastante interessantes e úteis.

Dando nomes aos canos

Um outro tipo de pipe é o *named pipe*, que também é chamado de FIFO. FIFO é um acrônimo de *First In First Out* que se refere à propriedade em que a ordem dos bytes entrando no pipe é a mesma que a da saída. O *name* em named pipe é, na verdade, o nome de um arquivo. Os arquivos tipo named pipes são exibidos pelo comando *ls* como qualquer outro, com poucas diferenças, veja:

```
$ ls -l pipe1
prw-r--r--   1   julio   dipao   0 Jan 22 23:11 pipe1|
```

o `p` na coluna mais à esquerda indica que `fifo1` é um named pipe. O resto dos bits de controle de permissões, quem pode ler ou gravar o pipe, funcionam como um arquivo normal. Nos sistemas mais modernos uma barra vertical (`|`), ou pipe, no fim do nome do arquivo é outra dica e, nos sistemas LINUX, onde o `ls` pode exibir cores, o nome do arquivo é escrito em vermelho por padrão.

Nos sistemas mais antigos, os named pipes são criados pelo utilitário *mknod*, normalmente situado no diretório `/etc`. Nos sistemas mais modernos, a mesma tarefa é feita pelo *mkfifo*, que recebe um ou mais nomes como argumento e cria pipes com esses nomes. Por exemplo, para criar um named pipe com o nome `pipe1`, digite:

```
$ mkfifo pipe1
```

Como sempre, a melhor forma de mostrar como algo funciona é dando exemplos. Suponha que nós tenhamos criado o named pipe mostrado anteriormente. Vamos agora trabalhar com duas sessões ou dois consoles virtuais. Em um deles digite:

```
$ ls -l > pipe1
```

e em outro faça:

```
$ cat < pipe1
```

Voilà! A saída do comando executado no primeiro console foi exibida no segundo. Note que a ordem em que os comandos ocorreram não importa.

Se você prestou atenção, reparou que o primeiro comando executado parecia ter "pendurado". Isto acontece porque a outra ponta do pipe ainda não estava conectada, e então o sistema operacional suspendeu o primeiro processo até que o segundo "abrisse" o pipe. Para que um processo que usa pipe não fique em modo de espera, é necessário que em uma ponta do pipe haja um processo "falante" e na outra um "ouvinte". No exemplo anterior, o `ls` era o "falante" e o `cat` era o "ouvinte".

Um uso muito útil dos named pipes é permitir que programas sem nenhuma relação possam se comunicar entre si. Os named pipes também são usados para sincronizar processos, já que em um determinado ponto você pode colocar um processo para "ouvir" ou para "falar" em um determinado named pipe e ele daí só sairá se outro processo "falar" ou "ouvir" aquele pipe.

Você já deve ter notado que essa ferramenta é ótima para sincronizar processos e fazer bloqueio em arquivos de forma a evitar perda/corrupção de dados devido a atualizações simultâneas (a famosa *concorrência*). Vamos ver alguns exemplos para ilustrar estes casos.

Sincronização de processos

Suponha que você dispare paralelamente dois programas (processos), chamados *programa1* e *programa2*, cujos diagramas de blocos de suas rotinas são como mostrado na **tabela 1**. Os dois processos são disparados em paralelo e, no bloco 1 do programa1, as três classificações são disparadas da seguinte maneira:

```
for Arq in BigFile1 BigFile2 BigFile3
do
    if  sort $Arq
    then
        Manda=va
    else
        Manda=pare
        break
    fi
done
echo $Manda > pipe1
[ $Manda = pare ] &&
    {
    echo Erro durante a classificação dos arquivos
    exit 1
    }
…
```

Assim sendo, o comando `if` testa cada classificação que está sendo efetuada. Caso ocorra qualquer problema, as classificações seguintes serão abortadas, uma mensagem contendo a string `pare` é enviada pelo pipe1 e programa1 é descontinuado com código de saída sinalizando um encerramento anormal.

Enquanto o programa1 executava o seu primeiro bloco (as classificações), o programa2 executava o seu bloco 1, processando as suas rotinas de abertura e menu paralelamente ao programa1, ganhando dessa forma um bom tempo. O fragmento de código do programa2 a seguir mostra a transição do seu bloco 1 para o bloco 2:

```
OK=`cat pipe1`
if  [ $OK = va ]
then
    …
    Rotina de impressão
```

Tabela 1

	Programa1	Programa2
Bloco 1	Rotina de classificação de três grandes arquivos	Rotina de abertura e geração de menus
Bloco 2	Acertos finais e encerramento	Impressão dos dados classificados pelo programa 1

```
    …
else
    exit 1
fi
```

Após a execução de seu primeiro bloco, o programa2 passará a "ouvir" o pipe1, ficando parado até que as classificações do Programa1 terminem, testando a seguir a mensagem passada pelo pipe1 para decidir se os arquivos estão íntegros para serem impressos ou se o programa deverá ser descontinuado. Dessa forma é possível disparar programas de forma assíncrona e sincronizá-los quando necessário, ganhando bastante tempo de processamento.

Bloqueio de arquivos

Suponha que você tenha escrito um CGI (*Common Gateway Interface*) em Shell Script para contar quantos hits uma determinada URL recebe e a rotina de contagem está da seguinte maneira:

```
Hits="$(cat page.hits 2> /dev/null)" || Hits=0
echo $((Hits=Hits++)) > page.hits
```

Dessa forma, se a página receber dois ou mais acessos simultâneos, um ou mais poderá ser perdido, bastando que o segundo acesso seja feito após a leitura do arquivo `page.hits` e antes da sua gravação, isto é, após o primeiro acesso ter executado a primeira linha do script e antes de executar a segunda.

Listagem 2: contahits.sh

```
01 #!/bin/bash
02
03 PIPE="/tmp/pipe_contador" # arquivo named pipe
04 # dir onde serao colocados os arquivos contadores de cada pagina
05 DIR="/var/www/contador"
06
07 [ -p "$PIPE" ] || mkfifo "$PIPE"
08
09 while :
10 do
11     for URL in $(cat < $PIPE)
12     do
13         FILE="$DIR/$(echo $URL | sed 's,.*/,,')"
14         # quando rodar como daemon comente a proxima linha
15         echo "arquivo = $FILE"
15
17         n="$(cat $FILE 2> /dev/null)" || n=0
18         echo $((n=n+1)) > "$FILE"
19     done
20 done
```

Então, o que fazer? Para resolver o problema de concorrência, vamos utilizar um named pipe. Criamos o script na **listagem 2** que será o daemon que receberá todos os pedidos para incrementar o contador. Note que ele vai ser usado por qualquer página no nosso site que precise de um contador. Como apenas este script altera os arquivos, não existe o problema de concorrência.

Este script será um *daemon*, isto é, rodará em segundo plano. Quando uma página sofrer um acesso, ela escreverá a sua URL no pipe. Para testar, execute este comando:

```
echo "teste_pagina.html" > /tmp/pipe_contador
```

Para evitar erros, em cada página a que quisermos adicionar o contador acrescentamos a seguinte linha:

```
<!--#exec cmd="echo $REQUEST_URI > /tmp/pipe_contador"-->
```

Note que a variável `$REQUEST_URI` contém o nome do arquivo que o browser requisitou. Esse exemplo é fruto de uma troca de idéias com o amigo e mestre em Shell Thobias Salazar Trevisan, que escreveu o script e colocou-o em seu excelente site (www.thobias.org). Aconselho a todos os que querem aprender Shell a dar uma passada lá.

Você pensa que o assunto named pipes está esgotado? Enganou-se. Vou mostrar um uso diferente a partir de agora.

Substituição de processos

Vou mostrar que o Shell também usa os named pipes de uma maneira bastante singular, que é a substituição de processos (*process substitution*). Uma substituição de processos ocorre quando você põe um < ou um > grudado na frente do parêntese da esquerda. Digitar o comando:

```
$ cat <(ls -l)
```

Resultará no comando `ls -l` executado em um sub-shell, como normal, porém redirecionará a saída para um named pipe temporário, que o Shell cria, nomeia e depois remove. Então o `cat` terá um nome de arquivo válido para ler (que será este named pipe e cujo dispositivo lógico associado é `/dev/fd/63`). O resultado é a mesma saída que a gerada pelo `ls -l`, porém dando um ou mais passos que o usual. Pra que simplificar?

Como poderemos nos certificar disso? Fácil… Veja o comando a seguir:

```
$ ls -l >(cat)
l-wx-- 1 jneves jneves 64 Aug 27 12:26 /dev/fd/63 -> pipe:[7050]
```

É... Realmente é um named pipe. Você deve estar pensando que isto é uma maluquice de nerd, né? Então suponha que você tenha dois diretórios, chamados `dir` e `dir.bkp`, e deseja saber se os dois são iguais. Basta comparar o conteúdo dos diretórios com o comando *cmp*:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) || echo backup furado
```

ou, melhor ainda:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) >/dev/null || echo backup furado
```

Este é um exemplo meramente didático, mas são tantos os comandos que produzem mais de uma linha de saída que ele serve como guia para outros. Eu quero gerar uma listagem dos meus arquivos, numerando-os, e ao final mostrar o total de arquivos no diretório corrente:

```
while read arq
do
    ((i++)) # assim nao eh necessario inicializar i
    echo "$i: $arq"
done < <(ls)
echo "No diretorio corrente (`pwd`) existem $i arquivos"
```

Tá legal, eu sei que existem outras formas de executar a mesma tarefa. Usando o comando while, a forma mais comum de resolver esse problema seria:

```
ls | while read arq
do
    ((i++)) # assim nao eh necessario inicializar i
    echo "$i: $arq"
done
echo "No diretorio corrente (`pwd`) existem $i arquivos"
```

Ao executar o script, tudo parece estar bem, porém no comando `echo` após o `done`, você verá que o valor de `$i` foi perdido. Isso deve-se ao fato desta variável estar sendo incrementada em um sub-shell criado pelo pipe (`|`) e que terminou no comando `done`, levando com ele todas as variáveis criadas no seu interior e as alterações lá feitas por variáveis criadas externamente.

Somente para te mostrar que uma variável criada fora do sub-shell e alterada em seu interior perde as alterações feitas quando o sub-shell se encerra, execute o script a seguir:

```
#!/bin/bash
LIST=""                 # Criada no shell principal
ls | while read FILE    # Inicio do subshell
do
    LIST="$FILE $LIST"  # Alterada dentro do subshell
done                    # Fim do subshell
echo $LIST
```

No início deste exemplo eu disse que ele era meramente didático porque existem formas melhores de fazer a mesma tarefa. Veja só estas duas:

```
$ ls | ln
```

ou então, usando a própria substituição de processos:

```
$ cat -n <(ls)
```

Um último exemplo: você deseja comparar `arq1` e `arq2` usando o comando `comm`, mas esse comando necessita que os arquivos estejam classificados. Então a melhor forma de proceder é:

```
$ comm <(sort arq1) <(sort arq2)
```

Essa forma evita que você faça as seguintes operações:

```
$ sort arq1 > /tmp/sort1
$ sort arq2 > /tmp/sort2
$ comm /tmp/sort1 /tmp/sort2
$ rm -f /tmp/sort1 /tmp/sort2
```

Pessoal, o nosso papo de botequim chegou ao fim. Curti muito e recebi diversos elogios pelo trabalho desenvolvido ao longo de doze meses e, o melhor de tudo, fiz muitas amizades e tomei muitos chopes de graça com os leitores que encontrei pelos congressos e palestras que ando fazendo pelo nosso querido Brasil. Me despeço de todos mandando um grande abraço aos barbudos e beijos às meninas e agradecendo os mais de 100 emails que recebi, todos elogiosos e devidamente respondidos. À saúde de todos nós: Tim, Tim.

- Chico, fecha a minha conta porque vou pra casa! ■

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. É autor do livro Programação Shell Linux, *publicado pela editora Brasport. Pode ser contatado no e-mail julio.neves@gmail.com*