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 $used_up
22 echo "O que sobrou da linha de comandos foi '$*'"
```