



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 $LinhaMsg $Col
10 read -n1 -p "$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 $LinhaMsg $Col
11 read -n1 -p "$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 $LinhaMsg $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 caractere % 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-barras (\). 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 caractere"
1$
```

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

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

Saltou linha mas ainda não listou a cadeia inteira

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

Listagem 2: listaartista

```
$ cat listaartista3.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 (<i>float</i>)
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

Tabela 2: Sequências de escape

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

(%) significa o tamanho que a cadeia terá após a execução do comando. Vamos ver a seguir mais alguns exemplos. Os comandos abaixo:

```
$ 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 experimentalmente:

```
$ 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 BOBO
```

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

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

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

O primeiro %X converteu 49354 em hexadecimal, resultando em COCA (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 sabia como enumerar tantas regrihas 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 frequência 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 ./script para que o script script seja executado. Basta digitar script. 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 “Papô 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="Papó 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
<code>\${var:-padrao}</code>	Se <code>var</code> é vazia, o resultado da expressão é padrão
<code>\${#cadeia}</code>	Tamanho de <code>\$cadeia</code>
<code>\${cadeia:posicao}</code>	Extrai uma subcadeia de <code>\$cadeia</code> a partir de posição. Origem zero
<code>\${cadeia:posicao:tamanho}</code>	Extrai uma subcadeia de <code>\$cadeia</code> a partir de posição com tamanho igual a <code>tamanho</code> . Origem zero
<code>\${cadeia#expr}</code>	Corta a menor ocorrência de <code>\$cadeia</code> à esquerda da expressão <code>expr</code>
<code>\${cadeia##expr}</code>	Corta a maior ocorrência de <code>\$cadeia</code> à esquerda da expressão <code>expr</code>
<code>\${cadeia%expr}</code>	Corta a menor ocorrência de <code>\$cadeia</code> à direita da expressão <code>expr</code>
<code>\${cadeia%%expr}</code>	Corta a maior ocorrência de <code>\$cadeia</code> à direita da expressão <code>expr</code>
<code>\${cadeia/subcad1/subcad2}</code>	Troca a primeira ocorrência de <code>subcad1</code> por <code>subcad2</code>
<code>\${cadeia//subcad1/subcad2}</code>	Troca todas as ocorrências de <code>subcad1</code> por <code>subcad2</code>
<code>\${cadeia/#subcad1/subcad2}</code>	Se <code>subcad1</code> combina com o início de cadeia, então é trocado por <code>subcad2</code>
<code>\${cadeia/%subcad1/subcad2}</code>	Se <code>subcad1</code> combina com o fim de cadeia, então é trocado por <code>subcad2</code>

```
$ echo $cadeia
Papode Botequim
$ echo ${cadeia%' '*}
Papode
$ echo ${cadeia%%%' '*}
Papode
```

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

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

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

```
$ echo $cadeia
Papode 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 todas 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