Colocar para 11a. edição

OBS para o revisor: É importante que as aspas e os apóstrofos sejam o que estou usando. Alguns editores trocam por *default* esses sinais, pelo que ele chamam de aspas e apóstrofos franceses. certo (" " ' '), errado (“ ” ‘ ’).

==========

Abaixo dos agradecimentos, colocar o título (com o mesmo estilo do Agradecimentos):

Dedicatória desta edição

e o texto:

Dedico esta edição ao Marcos Vinícius Mazoni, grande administrador (o melhor com o qual já trabalhei), com inteligência aguda, sagaz e brilhante.

Mazoni é um exemplo de honestidade de propósitos e de vida, com o qual tive o prazer e honra de trabalhar por 8 anos consecutivos.

==========

No final do prefácio da 9° edição faltou colocar o nome de quem a escreveu. Favor incluir:

Rubens Queiroz de Almeida
Trabalha no Centro de Computação da Unicamp desde 1988.
Mantém diversas páginas na Web, entre os quais o **Dicas-L**, (http://www.Dicas-L.unicamp.br), que foi o site pioneiro na abordagem do Linux em língua portuguesa.

==========

Na página 37, trocar:

2. Passando o conteúdo ...

```
$ wc < cat texto.txt
346      282      26
```

Por:

2. Passando o conteúdo ...

```
$ wc < texto.txt
346      282      26
```

==========

No fim da página 42, colocar um título com hierarquia superior ao "Tail - Mostra final dos dados"

Formatando dados na saída

Em seguida, inserir a seção a seguir com o título do mesmo estilo do Tail - Mostra final dos dados"

pr - converte textos para imprimir

Sintaxe:

pr [-opções] [ARQUIVO]

O comando **pr** permite que você formate **ARQUIVO** para impressão ou para listá-lo na tela. Mostrarei somente as opções mais úteis e sempre com saída na tela.

Opções	Significado
-NUM	Tabelar a saída em **NUM** colunas
-w NUM	A saída será formatada para ter largura **NUM**
-T	Omite a paginação. Sempre uso essa opção para formatar a saída na tela

Exemplos:

O nosso arquivo praias tem:

```
$ wc -l < praias
41
```

Esta foram consideradas as 41 praias mais lindas do Brasil (as que já conheci, são realmente para não deixar nada devendo às Bahamas).

Como a listagem dessas lindezas ficaria muito extenso na tela, uso o comando a seguir para listá-las:

```
$ pr -T -w80 -3 praias
```

Que produz a saída a seguir:

```
$ pr -T -w80 -3 praias
Angra dos Reis, RJ         Praia Central de Camboriú,  Praia do Farol, RJ
Barra do Sahy, SP          Praia da Barra, RJ          Praia do Forte, BA
Boa Viagem, PE             Praia da Joaquina, SC       Praia do Futuro, CE
Copacabana, RJ             Praia da Pipa, RN           Praia do Rosa, SC
Garopaba, SC               Praia das Conchas, RJ       Praia do Sancho, Fern. Nor
Geribá, RJ                 Praia de Grumari, RJ        Praia dos Dois Rios, RJ
Grumari, RJ                Praia de Ipanema, RJ        Praia José Gonçalves, RJ
Itacaré, BA                Praia de Ipanema, RJ        Prainha, RJ
Jericoacoara, CE           Praia de Itacoatiara, RJ    Prainhas do Pontal, RJ
Jurerê, SC                 Praia de Itapuã, BA         Refúgio, SE
Pajuçara, AL               Praia de Lopes Mendes, RJ   Taipus de Fora, BA
Porto da Barra, BA         Praia de Ponta Negra, RN    Tambaú, PB
Porto de Galinhas, PE      Praia de São Marcos, MA      Trancoso, BA
Praia Azeda e Azedinha, RJ Praia do Espelho, BA
$
```

Neste exemplo, as praias foram divididas em 3 colunas **(-3)** com 80 de largura **(-w80)**, que é menos que o necessário, você pode notar isso logo na 1ª. linha, que a UF

da **Praia Central** foi omitida. Veja também que na **Praia do Sancho, Fern. Noronha** foi truncado,

Caso não houvesse usado a opção **-T,** ao fim de cada página seria inserido um *form feed* e seria criado o cabeçalho do tipo deste a seguir:

```
2016-02-29 08:59                    praias                    Página 1
```

==========

Continuando, inserir a seção a seguir, com título do mesmo estilo do anterior:

column - criando colunas na tela

Sintaxe:

column [-opções] [ARQUIVO]

O comando **column** não se preocupa em gerar relatórios, sua finalidade é tabular os dados que vão para tela. Suas principais opções são:

A opção **-t** formata uma tabela para impressão, na forma que considera ser o ideal.

A opção **-c NUM** monta a saída para uma tela, cuja largura disponibilizada para fazer a tabulação seja **NUM.**

A opção **-s,** quando usada junto à **-t,** explicita qual á o separador entre campos da entrada. Muito útil quando usado com arquivos com campos separados por vírgulas **(.csv).**

Observação: a opção **-s** aceita múltiplos separadores.

Um exemplo de uso da opção **-t** com tabulação automática:

```
$ (echo PERMISSOES LNK DONO GRUPO TAM MES DIA \
HH:MM NOME-PROG; ls -l | tail -n +2) | column -t
PERMISSOES  LNK  DONO   GRUPO  TAM   MES  DIA  HH:MM  NOME-PROG
drwxr-xr-x  2    julio  julio  4096  Mar  7    09:51  dir
drwxr-xr-x  2    julio  julio  4096  Mar  11   09:00  dir1
-rw-r--r--  1    julio  julio  10    Mar  8    15:24  impar
-rw-r--r--  1    julio  julio  11    Mar  8    15:24  pars.txt
```

Nesse exemplo os parênteses criam um *subshell* para executar um bloco de comandos. A saída desse bloco será mandado para o comando **column.** Nesse bloco, primeiramente montamos um cabeçalho e abaixo dele a saída do comando **ls -l,** cuja primeira linha mostra sempre o total de blocos usados no diretório que está sendo listado e que não interessa para o nosso caso e, por isso, foi excluída pelo comando **tail.**

Usando essa opção, a tabulação foi feita colocando 2 espaços em branco após o maior elemento de cada coluna. Repare que na coluna **DONO,** o maior elemento é **julio,** então a próxima coluna fica afastada 2 espaços da palavra **Julio,** já na coluna **TAM** o maior é **4096,** então a coluna **MES** estará 2 espaços após o **4096.**

Exemplos de uso da opção **-c NUM:**

Destinando 90 colunas para listar o arquivo **praias (column -c90 praias)**:

```
$ column -c90 praias
Angra dos Reis, RJ                Praia de Ipanema, RJ
Barra do Sahy, SP                 Praia de Itacoatiara, RJ
Boa Viagem, PE                    Praia de Itapuã, BA
Copacabana, RJ                    Praia de Lopes Mendes, RJ
Garopaba, SC                      Praia de Ponta Negra, RN
Geribá, RJ                        Praia de São Marcos, MA
Grumari, RJ                       Praia do Espelho, BA
Itacaré, BA                       Praia do Farol, RJ
Jericoacoara, CE                  Praia do Forte, BA
Jurerê, SC                        Praia do Futuro, CE
Pajuçara, AL                      Praia do Rosa, SC
Porto da Barra, BA                Praia do Sancho, Fern. Noronha
Porto de Galinhas, PE             Praia dos Dois Rios, RJ
Praia Azeda e Azedinha, RJ        Praia José Gonçalves, RJ
Praia Central de Camboriú, SC     Prainha, RJ
Praia da Barra, RJ                Prainhas do Pontal, RJ
Praia da Joaquina, SC             Refúgio, SE
Praia da Pipa, RN                 Taipus de Fora, BA
Praia das Conchas, RJ             Tambaú, PB
Praia de Grumari, RJ              Trancoso, BA
Praia de Ipanema, RJ
```

Destinando 110 colunas para listar o arquivo **praias (column -c110 praias)**:

```
$ column -c110 praias
Angra dos Reis, RJ              Praia Central de Camboriú, SC   Praia do Farol, RJ
Barra do Sahy, SP               Praia da Barra, RJ              Praia do Forte, BA
Boa Viagem, PE                  Praia da Joaquina, SC           Praia do Futuro, CE
Copacabana, RJ                  Praia da Pipa, RN               Praia do Rosa, SC
Garopaba, SC                    Praia das Conchas, RJ           Praia do Sancho, Fern. Noronha
Geribá, RJ                      Praia de Grumari, RJ            Praia dos Dois Rios, RJ
Grumari, RJ                     Praia de Ipanema, RJ            Praia José Gonçalves, RJ
Itacaré, BA                     Praia de Ipanema, RJ            Prainha, RJ
Jericoacoara, CE                Praia de Itacoatiara, RJ        Prainhas do Pontal, RJ
Jurerê, SC                      Praia de Itapuã, BA             Refúgio, SE
Pajuçara, AL                    Praia de Lopes Mendes, RJ       Taipus de Fora, BA
Porto da Barra, BA              Praia de Ponta Negra, RN        Tambaú, PB
Porto de Galinhas, PE           Praia de São Marcos, MA         Trancoso, BA
Praia Azeda e Azedinha, RJ      Praia do Espelho, BA
```

Um exemplo de uso da opção **-s**:

Primeiramente vamos criar uma entrada separada por vírgulas **(,)**:

```
$ ls | paste -d, - - -
anda.sh,animais,AnoNovo.sh
anosmeses,arq,ARQ
arq1,arq2,arq3.ok
arq.err,arqr,Arqs
ArteAscii2.sh,ascii,ascii1
asd,Astro1.png,aux
aux1,aux2,b
Banner.jpg,bb,bbb
besteira.txt,Caipirinha.png,calc10.sh
calc10.yad,calc1.yad,calc.sh
calculadora.yad,calcvictor.yad,calcv.yad
```

Os traços à frente do `paste` recebem os campos da entrada primária (*stdin*) passados pelo *pipe* `(|)` e coloca vírgulas `(-s,)` entre cada um deles, como vimos.

Vamos então coluná-lo:

```
$ ls | paste -d, - - - | column -ts,
anda.sh                 animais           AnoNovo.sh
anosmeses               arq               ARQ
arq1                    arq2              arq3.ok
arq.err                 arqr              Arqs
ArteAscii2.sh           ascii             ascii1
besteira.txt            Caipirinha.png    calc10.sh
calc10.yad              calc1.yad         calc.sh
calculadora.yad         calcvictor.yad    calcv.yad
```

Um exemplinho bem bobo, só para mostrar o uso com diversos separadores:

```
$ echo 1-2^3@4 | column -ts@-^
1  2  3  4
```

==========

Trocar a tabela da pág 69, pela seguinte:

Opções	Significados
`-e`	Apresenta a linha de comando completa
`-l`	Gera saída em formato longo
`-a`	Inclui informações referentes aos processos de todos os usuários
`-u`	Produz saída orientada a usuários
`-x`	Inclui processos não associados a terminais
`-oLISTA`	`LISTA` especifica os dados (colunas) que constaram na saída [1]
`--sort=(+|-)LISTA`	Diz que a saída será ordenada por cada um dos elementos de `LISTA`. O sinal `+` é *default*, mas o menos indica uma ordenação decrescente [1]

[1] Nesses 2 últimos casos, para construir `LISTA` (que tem seus componentes separados por vírgulas), costumo retirar os dados do cabeçalho de `ps aux`, veja:

```
$ ps aux | head -1
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
```

Nessa linha, o `head -1` pega a 1ª linha (cabeçalho) da saída do comando `ps`

==========

Na pág 70, Inserir antes do título Kill ...

Repare que se usarmos a opção `-o,` além de especificar as colunas, podemos também alterar o cabeçalho da saída simplesmente usando-se um sinal de igual `(=)`.

```
$ ps -o pid,ruser=UsuárioReal -o comm=Comando
  PID UsuárioReal Comando
 2419 julio       bash
11089 julio       ps
```

Como você viu, além de especificarmos as colunas da saída, ainda alteramos o cabeçalho.

Veja mais esse:

```
$ ps -xo uid,pid,ppid,comm --sort=uid,-ppid,pid
  UID   PID  PPID COMMAND
  ...   ...  ...  ...
  ...   ...  ...  ...
```

Neste último exemplo em que usamos reticências para substituir o texto, a opção `-x` servia para listar todos os processos em execução, o `-o` para especificar as colunas da saída e o `--sort` estipulou que a saída seria ordenada por `UID,` em seguida, em ordem decrescente, causada pelo sinal de menos `(-),` o `PPID` e finalmente pelo `PID` (que poderia ter um sinal de mais `(+)` à frente mas que não foi usado por ser o padrão).

Na pág 92, antes do título Passa linha de comando para o kernel, inserir:

Vejamos alguns exemplos de como o *Shell* faz com que a substituição de metacaracteres se expanda para nome de arquivos:

```
$ ls *                          Lista todos os arquivos do diretório
arq1  arq12  arq2  arq3  ximba3.c  xoops.txt
$ ls arq
ls: não é possível acessar arq: Arquivo ou diretório não encontrado
$ ls arq*                       Arquivos começados por arq
arq1  arq12  arq2  arq3
$ ls arq?                       Idem anterior, porém com arq + 1 caractere
arq1  arq2  arq3
$ ls arq??                      arq seguido de 2 caracteres
arq12
$ ls arq[12]                    arq seguido por 1 ou 2
arq1  arq2
$ ls arq[!12]                   arq seguido de 1 caractere, que não seja 1 ou 2
arq3
$ ls arq[^12]                   Também negando, mas com outra sintaxe
arq3
$ ls *.{txt,c}                  Expande o prefixo (*) e o combina com txt e c
ximba3.c  xoops.txt
```

Assim podemos ver que o asterisco `(*)` casa com tudo, o ponto de interrogação `(?)` casa com somente um caractere, seja ele qual for, a lista `([...])` também casa com somente um caractere, porém um que tenha sido especificado e a lista negada `([!...]` ou `[^...])` casa com qualquer caractere, exceto os especificados.

O último exemplo é uma expansão de chaves, cuja explicação mais detalhada será vista mais à frente, mas já adianto que serve para combinar o prefixo `(*.)` com cada elemento no interior das chaves e com o sufixo (que não foi usado nesse caso).

Poderíamos ter usado o comando `echo` no lugar do `ls`, obtendo o mesmo resultado, porém com uma formatação diferente. Isso serve para demonstrar que o *Shell* expande os metacaracteres, antes de mandar o comando para execução.

Uma exceção: nenhum metacaractere expande para o ponto (.) nos arquivos escondidos (os que começam por ponto), a não ser que se inclua explicitamente o ponto como um caractere literal. Portanto cuidado quando for pesquisar um estes arquivos, veja:

```
$ ls ~/.profile
/home/julio/.profile            .profile existe no home
$ ls ~/.pr*
/home/julio/.profile            Com o ponto explícito, casa. Senão estiver explícito ...
$ ls ~/?profile
ls: /home/julio/?profile: Arquivo ou diretório não encontrado
$ ls ~/*profile
ls: /home/julio/*profile: Arquivo ou diretório não encontrado
$ ls ~/[.]profile
ls: /home/julio/[.]profile: Arquivo ou diretório não encontrado
$ ls ~/[!a]profile
ls: /home/julio/[!a]profile: Arquivo ou diretório não encontrado
```

==========

Na 2ª. tabela da pag 101, incluir como sua última linha:

`<<<` Também conhecido como *here strings*, substitui a dupla `echo ESCOPO | COMANDO`, que é muito usada, mas cria *subshell*.

==========

Antes dos exercícios da pag 103, incluir:

Não sei porque, mas é muito raro vermos alguém utilizando de *here strings*. Seu uso deve ser fortemente incentivado, já que evita usarmos *pipes*, que fazem *fork* no seu *script*, criando *subshells*. Veja isso:

```
$ a=bicicleta
$ echo 2+2 | {                  Pipe vai rodar um bloco de cmds num subshell
> bc
> a=motocicleta                 Mudei valor de $a no subhell gerado pelo pipe
> echo $a
> }
4                               Saida do comando bc
motocicleta                     Saída do $a dentro do subshell
$ echo $a
bicicleta                       Fora do subshell o valor de $a não foi alterado
```

Como este exemplo demonstrou, usar um *subshell* além de ser um pouco mais lento, pode te levar a se confundir. A melhor forma de passar uma conta para o `bc` fazer é:

```
$ bc <<< 2+2
4
```

Esse redirecionamento, é muito importante e mais à frente, volto a abordá-lo com mais profundidade. Por enquanto basta você saber (e aplicar) que todo `echo ESCOPO | CMD` onde `CMD` é qualquer comando, deve ser substituído por `CMD <<< ESCOPO`

==========

Na página 105, abaixo do título do capítulo, como se fosse um adendo deste, porém com a fonte de Corpo do Texto normal (*Bitstream Vera Sans* 12px?) incluir:

(O Itamar Santos de Souza, PhD em assuntos "sedianos e awkdianos" e do qual vocês breve terão notícias, muito me ajudou nesse capítulo, revisando-o e sugerindo dicas. Valeu Itamar!)

==========

Pág 108, substituir a partir do 1º. parágrafo, onde está: de duas formas:, até `NINI,/CAD2/` por:

de várias formas:

O basicão:

`/EXPR1/,/EXPR2/`	*Onde EXPR pode ser cadeia ou expressão regular*
`NINI,NFIM`	*Número Inicial e Final*
`/CAD1/,+NUM`	*Da cadeia CAD1 até NUM linhas após*
`NINI, +NUM`	*Da linha número NINI até NUM linhas após*

Um pouquinho mais incrementado:

`/CAD1/,~NUM`	*Da cadeia CAD1 até próxima linha múltipla de NUM*
`NINI, ~NUM`	*Da linha NINI até próxima linha múltipla de NUM*
`NINI~INCR`	*Da linha NINI, incrementando de INCR em INCR*

Alguns exemplos:

```
$ seq 10 | sed -n '/2/,/^4$/p'
2
3
4
```

Neste caso, o `2` entre barras, estava sendo visto como a cadeia `2` e o `4` era uma expressão regular que dizia que entre o início `(^)` e o fim `($)` só existia a cadeia `4`.

ATENÇÃO!

ATENÇÃO! É muito importante notar que o cifrão `($)` significa fim, então se ele for tratado como número (sem estar entre barras) significa fim do arquivo, ou seja, a última linha, entre barras é a expressão regular que representa fim da linha. Assim, para deletar a última linha, basta fazer:

```
$ seq 3 | sed '$d'
1
2
```

Para listar da segunda linha até a quarta, também podemos fazer:

```
$ seq 10 | sed -n '2,4 p'
2
3
4
```

Outra forma de fazer a mesma coisa é com incremento. Então para listar da segunda até a segunda `+2`, isto é, quarta, também podemos fazer:

```
$ seq 10 | sed -n '2,+2 p'
2
3
```

```
4
```

O exemplo a seguir, lista desde a linha que tem a cadeia `2` até a segunda após esta:

```
$ seq 10 | sed -n '/2/,+2 p'
2
3
4
```

Da oitava linha até a próxima linha de ordem múltipla de `3:`

```
$ seq 10 | sed -n '8,~3 p'
8
9
```

Idem em linha de ordem múltipla de `2:`

```
$ seq 10 | sed -n '8,~2 p'
8
9
10
```

Nestes dois últimos exemplos, não procurei pela linha que tivesse um `9` ou um `10,` listei a partir da linha 8 até a próxima linha cuja ordem fosse múltipla de `3` e logo após pela múltipla de `2.` Para não fazer confusão veja o mesmo exemplo porém usando letras:

```
$ echo {A..E} | tr ' ' '\n'
A
B
C
D
E
$ echo {A..E} | tr ' ' '\n' | sed -n '2,~3 p'
B
C
$ echo {A..E} | tr ' ' '\n' | sed -n '2,~2 p'
B
C
D
```

Para listar somente as linhas de ordem par, com início em `2` e incremento de `2:`

```
$ seq 10 | sed -n '2~2p'
2
4
6
8
10
```

Para as linhas de ordem ímpar inicio em `1` e também uso incremento de `2:`

```
$ echo {A..E} | tr ' ' '\n' | sed -n '1~2 p'
A
C
E
```

==========

Na página 113, antes de Vamos dar um rewind... Inserir:

Para criarmos uma linha após a última basta fazer:

```
$ seq 5 | sed '$a \
6'
1
2
3
4
5
6
```

==========

Na página 115, antes da seção Abortando... inserir:

Veja só:

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

Nesse `^M$` ao final, o `^M` significa um *carriage return* e o `$` significa o fim (como sempre) da linha. Então poderíamos excluir o *carriage return*, cujo valor hexadecimal é `0d`, da seguinte maneira:

```
$ sed 's/\x0d//' DOS.txt | cat -vet
Este arquivo$
foi gerado pelo$
DOS/rwin e foi$
baixado por um$
ftp mal feito.$
```

Como o valor ascii do *carriage return* em decimal é `13` e em octal é `15`, poderíamos fazer o mesmo de qualquer uma das duas formas a seguir:

```
$ sed 's/\d13//' DOS.txt          \d define decimal
```

ou:

```
$ sed 's/\o15//' DOS.txt          \o define octal
```

Uma outra forma de fazer o mesmo seria:

```
$ sed 's/\cM//' < DOS.txt | cat -vet
```

Desta forma, prefixamos uma letra com um `\c` para dizer que o caractere seguinte é um caractere de controle. Assim podemos representar o *carriage return*, que se apresenta como `(^M - CONTROL M)` com um `\cM` e o `<TAB>`, que se apresenta como `(^I - CONTROL I)` com um `\cI`.

Como o *carriage return* também pode ser representado por `\r` (veja as *escape sequences* na seção sobre o comando `tr`) também poderíamos fazer:

```
$ sed 's/\r//' DOS.txt
```

==========

Na página 120, antes do título Dicas sobre delimitadores, incluir:

Podemos também capitalizar (primeira letra de cada palavra em maiúscula e o resto

em minúscula) um texto:

```
$ cat nomes.txt
alice ada
eva dias
eça fadoa bessa
ivana pita
maria berta bessa
paulo botelho carvalho
peter punk
$ sed 's/\b\([[:alpha:]]\+\)\b/\u\1\L/g' nomes.txt
Alice Ada
Eva Dias
Eça Fadoa Bessa
Ivana Pita
Maria Berta Bessa
Paulo Botelho Carvalho
Peter Punk
```

Ou usando a opção `-r` da qual acabamos de falar, mas que ainda não foi explicada:

```
$ sed -r 's/\b([[:alpha:]]+)\b/\u\1\L/g' nomes.txt
```

Observação: caso os nomes no arquivo estivessem todos escritos em letras maiúsculas, o resultado seria o mesmo.

==========

Na página 128, antes do título Inserindo dados de outro arquivo, inserir:

É muito interessante notar que as *flags* de leitura e gravação externas `(/w` e `/r)` precisam ser as últimas nos comandos e caso precise usar comandos adicionais ou usar ambas as *flags*, será necessário usar a opção `-e` para separar cada trecho do *script*.

==========

Página 130 trocar:

```
sed 's/^\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3\/\2\/\1/'
<<< 31/12/2009
2009/12/31
```

Por:

```
sed 's/^\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3\/\2\/\1/' \
<<< 31/12/2009
2009/12/31
```

Trocar:

```
$ sed -r 's/^([0-9]{2})\/([0-9]{2})\/([0-9]{4})$/\3\/\2\/\1/'
<<< 31/12/2009
2009/12/31
```

Por:

```
$ sed -r 's/^([0-9]{2})\/([0-9]{2})\/([0-9]{4})$/\3\/\2\/\1/' \
<<< 31/12/2009
2009/12/31
```

==========

Pag 135, antes da seção Evitando o pipe, incluir outra seção com o título no mesmo estilo:

A opção -s

Se você estiver editando um monte de arquivos e está fazendo as mesmas coisas em todos, use a opção `-s`.

Digamos que a faculdade que você trabalha, tenha um arquivo de notas para cada matéria. Suponha também que houve greve em março e nenhuma prova foi aplicada. Para deletar a 3º linha (referente à 3º nota, isto é, março) de cada matéria podemos fazer:

```
$ sed -s '3d' algoritimos compiladores programacaoI
```

Sem o uso da opção `-s` apenas a terceira linha do arquivo `algoritimos` será apagada.

Essa opção é interessante pois evita o uso de um *loop* para fazer o trabalho todo.

==========

Na página 138, último parágrafo, trocar: Existem oito comandos por: Existem alguns comandos

==========

No último exemplo da pag 140, incluir um ponto e vírgula quase no final trocando:

```
$ seq 6 | sed '/3/{d;n}'
```

por:

```
$ seq 6 | sed '/3/{d;n;}'
```

Observação: o ponto e vírgula `(;)` após o `n` é obrigatório somente para ambientes BSDs

==========

Na pag 146 antes do título Desvio Incondicional, inserir:

Como acabamos de ver, o comando `t` redireciona o fluxo do programa baseado no sucesso de uma substituição, mas ao contrário desta, também existe (só no GNU sed?) o comando `T ROTULO,` que retorna para `:ROTULO` se a substituição não tiver sucesso.

==========

No início da página 148, com o mesmo estilo do título da seção A família de comandos grep, Incluir o seguinte texto:

As vezes os navegadores ajudam

Mais uma tirada da lista s*hell-script* do Yahoo. A pessoa tinha a seguinte dúvida:

Existe alguma forma de exportar um dado de origem HTML para TXT via *shell script*? Eu tenho diversas linhas mais ou menos assim:

```
<tr><td><strong>Tensão de entrada:</strong></td><td>218.9 V</td></tr>
```

E dessa linha só me interessa o `218.9`. Como fazer isso?

E o Itamar Santos de Souza, um colega da lista que sabe tudo e mais um pouco sobre

`sed` e `awk`, respondeu:

"Pode usar os navegadores modo texto, e depois filtrar com `sed`, `awk`, `cut`, etc. Por exemplo:"

```
echo "<tr><td><strong>Tensão de entrada: \
    </strong></td><td>218.9 V</td></tr>" \
    | links -dump
```

ou:

```
echo "<tr><td><strong>Tensão de entrada: \
    </strong></td><td>218.9 V</td></tr>" \
    | lynx -dump -stdin
```

ou ainda:

```
echo "<tr><td><strong>Tensão de entrada: \
    </strong></td><td>218.9 V</td></tr>" \
    | w3m -T text/html
```

E para tirar os números desejados bastaria concatenar qualquer um desses casos em um *pipe* com um simples `sed`

```
"Um dos 3 comandos anteriores" | sed 's/[^0-9.]//g'
```

Vamos analisar o segundo, já que todos fazem a mesma coisa e o `lynx` é o navegador modo texto mais usado e é o único deles que tenho instalado.

A opção:

`-dump` – Manda o documento formatado (pelo HTML) para a saída;

`-stdin` – Aceita que o documento a formatar venha da entrada padrão (no caso o *pipe*)

Então executando o comando vem:

```
$ echo "<tr><td><strong>Tensão de entrada:\
    </strong></td><td>218.9 V</td></tr>"  \
    | lynx -dump -stdin
    Tensão de entrada: 218.9 V
```

Pronto agora é só usar o `sed` para deixar somente o número que interessa:

```
$ echo "<tr><td><strong>Tensão de entrada:\
    </strong></td><td>218.9 V</td></tr>"  \
    | lynx -dump -stdin | sed 's/[^0-9.]//g'
218.9
```

==========

Na pag 154, trocar:

```
$ sed -i 's/^/\^/;s/$/:/' usus
```

por:

```
$ sed -i 's/^/^/;s/$/:/' usus
```

==========

Na pág 161, a partir de Para extrair somente os nomes: até a linha anterior ao título do paste, trocar por:

Para extrair somente os nomes:

```
$  cut -f1 telefones                    Delimitador <TAB> é o default. Não especifiquei
Ciro Grippi
Ney Gerhardt
Enio Cardoso
Claudia Marcia
Paula Duarte
Ney Garrafas
```

Quero somente o primeiro nome:

```
$ cut -f1 -d  telefones                 Depois do -d eu coloquei um espaço
cut: o delimitador deve ser um único caractere
Try 'cut --help' for more information.
```

ZEBRA!!!! Eu queria pegar o nome que estivesse antes do delimitador espaço mas esqueci que espaços, em *Shell*, devem estar entre aspas. Vamos tentar novamente:

```
$ cut -f1 -d " " telefones              O d " " indica que o delimitador é branco
Ciro
Ney
Enio
Claudia
Paula
Ney
```

Para extrair os códigos de DDD:

```
$ cut -f2 -d"(" telefones | cut -f1 -d")"
021
024
023
021
011
021
```

Neste exemplo, se não protegêssemos os parênteses da interpretação do *Shell* colocando-os entre aspas, olha só a encrenca:

```
$ cut -f2 -d( telefones | cut -f1 -d )
bash: erro de sintaxe próximo do `token' não esperado `('
```

Isso ocorreria porque os parênteses, como já vimos, agrupam comandos e os executam em um *subshell*, assim sendo, se não os protegermos da interpretação do *Shell*, ele tentará fazer isso.

Mais três opções do **cut** que apesar de pouco usadas, são muito útil porque em casos específicos evitam um grande trabalho. São elas:

-s	Não lista as linhas que não possuem o delimitador especificado;
--complement	Lista tudo menos o campo definido;
--output-demimiter	Especifica o delimitador que virá na saída do comando.

Veja os exemplos:

```
$ ls arq*
arq1  arq2.txt  arq3.sh
```

Para pegar somente as extensões (mas repare que `arq1` não tem extensão), devemos fazer:

```
$ ls arq* |cut -f2 -d.
arq1
txt
sh
```

EPA! `arq1` não tinha de estar aí! O que houve? O problema foi ocasionado pelo padrão (*default*) do `cut` que, caso o alvo não tenha o separador definido, manda tudo está para a saída. Para evitar que isso aconteça, ou seja, para termos como resposta somente os campos que especificamos, existe a opção `-s.` Veja ela em uso:

```
$ ls arq* |cut -sf2 -d.
txt
sh
```

Veja agora o uso das opções `--complement` e `--output-delimiter:`

```
$ seq -s: 10
1:2:3:4:5:6:7:8:9:10
$ seq -s: 10 | cut -f5 -d: --complement
1:2:3:4:6:7:8:9:10                    Não listou o 5º. campo
$ seq -s: 10 | cut -f5 -d: --complement --output-delimiter \|
1|2|3|4|6|7|8|9|10                    Subtituiu, na saída, o delimitador : por |
$ seq -s: 10 | cut -f5 -d: --complement --output-delimiter $'\n'
1
2
3
4
6
7
8
9
10
```

==========

Na tabela da pag 229 consertar os sobrescritos nas seguintes células:

`\>`[1]

`\<`[1]

`-a`[2]

`-o`[2]

`\( ... \)`[2]

Consertar também os sobrescritos nas linhas logo abaixo desta tabela, ficando:

[1] Esta é uma extensão do padrão POSIX; alguns *Shells* podem tê-la, e outros não.

[2] Os operadores `-a` e `-o,` e o agrupador `(...),` são definidos pelo padrão POSIX, mas apenas para casos estritamente limitados porque são marcados como obsoletos. O uso desses operadores é desencorajado e é melhor você usar vários comandos `[.`

==========

No último parágrafo da pág 238, trocar:

O operador lógico || obriga...

Por:

O operador lógico `||` obriga...

Repare que o estilo do `||` mudou

==========

Na página 239, 2º. parágrafo, trocar:

pode ser aberto por um `do,` por um `if,` por um `else` ou...

Por:

pode ser aberto por um `do,` por um `then` (dentro do comando `if`), por um `else` (também dentro de um `if`) ou...

==========

No final da página 246, tirar o quadro de Dicas no fim da página e em seu lugar escrever:

Quando se está trabalhando direto no *prompt* é sempre saudável salvar a variável `"$IFS"` pois só existem duas formas de recuperá-la:

1. Desconectar-se (*logout*) e conectar-se (*login*) novamente ou;
2. Se você estiver sob o Bash, pode também fazer:

   ```
   $ IFS=$' \t\n'
   ```

   O 1º. Caractere é um espaço em branco

   Só para verificar:

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

Achei um texto muito interessante sobre isso em http://bash.cyberciti.biz/guide/$IFS. Vou traduzi-lo e adaptá-lo.

Vamos criar um *script* chamado `ifsargs.sh:`

```
$ cat ifsargs.sh
#!/bin/bash

echo "Demo de argumentos de linha de comando"
echo "*** Exibindo todos argumentos com \$@ ***"
echo $@
echo "*** Exibindo todos argumentos com \$* ***"
echo $*
```

Execute-o assim:

```
$ ./ifsargs.sh honda yamaha harley-davidson kawasaki
Demo de argumentos de linha de comando
*** Exibindo todos argumentos com $@ ***
honda yamaha harley-davidson kawasaki
*** Exibindo todos argumentos com $* ***
honda yamaha harley-davidson kawasaki
```

Como você pode ver, os valores de `$*` e de `$@` são os mesmos, no entanto se as variáveis estivessem entre aspas, seus valores seriam diferentes. Edite o *script* para ficar assim:

```
$ cat ifsargs.sh
#!/bin/bash

#### Passando o IFS para | ####
IFS=\|                      # O separador na saída será |

echo "Demo de argumentos de linha de comando"
echo "*** Exibindo todos argumentos com \$@ ***"
echo "$@"                   # Agora com aspas
echo "*** Exibindo todos argumentos com \$* ***"
echo "$*"                   # Agora com aspas
```

Vamos executá-lo novamente:

```
$ ./ifsargs.sh honda yamaha harley-davidson kawasaki
Demo de argumentos de linha de comando
*** Exibindo todos argumentos com $@ ***
honda yamaha harley-davidson kawasaki
*** Exibindo todos argumentos com $* ***
honda|yamaha|harley-davidson|kawasaki
```

Como você viu:

`$*` expandiu para `"$1" "$2" "$3" ... "$n"`

`$*` expandiu para `"$1y$2y$3y...$n"`, onde `y` é o valor da variável `$IFS`, isto é, `"$*"` é uma cadeia longa e o `$IFS` atua como um separador da saída.

==========

Na página 259 cometi um erro e troquei o texto. Onde está escrito `tput setf` troque *background* por *foreground* e, na linha seguinte, onde está escrito `tput setb` troque por *foreground* por *background*

==========

A tabela que mandei na edição anterior para a página 260, era para ter o seguinte formato:

Cores do terminal	
Símbolo	**Cor**
0	Preto
1	Azul
2	Verde
3	Ciano
4	Vermelho
5	Magenta
6	Amarelo

7	Branco
9	Volta a cor *default*

==========

Incluir na pag 280, antes da seção Esquisitices do IFS

Opção -r (título com estilo idêntico ao anterior)

Esta opção não permite a contrabarra atuar como um caractere de *escape* (normalmente usada no par contrabarra-*newline* que permite múltiplas linhas no `read`). Ela é considerada como parte da linha. Sem essa opção as contrabarras da entrada seriam descartadas.

Quando você está usando *Bash*, você pode atribuir à variável `IFS` o valor do `<ENTER>` (*newline* ou *line feed*) e ler de um *here strings* `(<<<)`. Podemos também dividir o conteúdo de uma variável para um vetor e assim acessarmos individualmente cada elemento desse vetor em um *loop* (usando `"${a[@]}"`).

```
$echo "$Var"
a
b c
d
f
g
$ IFS=$'\n' read -d '' -r -a Vet <<< "$Var"
$ echo "${Vet[0]}"
a
$ echo "${Vet[1]}"
b c
$ echo "${Vet[@]}"
a
b c
d
f
g
```

`read -d ''` Usa o um caractere *null* como delimitador (já vi essa representação como `IFS=$'\n' read -d $'\0' -r -a Vet <<< "$Var"`, onde `$'\0'` gera um caractere *null* ou um zero binário);

`-a Vet` Lê para o vetor `Vet;`

`-r` Esta opção não permite a contrabarra atuar como um caractere de *escape* (normalmente usada no par contrabarra-*newline* que permite múltiplas linhas no `read`). Ela é considerada como parte da linha. Sem essa opção as contrabarras da entrada seriam descartadas.

Outro exemplo mais palatável (fácil de engolir):

```
$ read var
a\                                  Com a contrabarra o Shell não viu o <ENTER>
b
$ echo $var
ab
$ read -r var                       Farei o mesmo que fiz no anterior
```

```
a\                                  Quando dei <ENTER> a leitura encerrou
$ echo $var
a\
$ read var
a\nb
$ echo $var
anb                                 O Shell tomou para si a contrabarra
$ read -r var
a\nb
$ echo $var
a\nb                                A contrabarra permaneceu
$ echo -e $var                      Com a opção -e o \n vira <ENTER>
a
b
```

Final da pág 286 a partir de `$ cat colunador.sh`, até ao 3º parágrafo, substituir por:

```
$ cat colunador.sh
#!/bin/bash
#  Recebe parâmetros via pipe ou passagem de
#+ parâmetros e os coloca em coluna numerando-os

(($# == 0)) && {  # Os dados vêm por pipe ou por parâmetro?
    Parms=$(cat -)
    set $Parms
}

for ((i=1; i<="$#"; i++))
{
    Lista=$(for ((i=1; i<="$#"; i++)); { printf "%0${##}i %s\n" $i ${!i};
})
}
echo "$Lista" | pr -T -8
$ echo {A..Z} | colunador.sh
01 A   05 E   09 I   12 L   15 O   18 R   21 U   24 X
02 B   06 F   10 J   13 M   16 P   19 S   22 V   25 Y
03 C   07 G   11 K   14 N   17 Q   20 T   23 W   26 Z
04 D   08 H
```

Esse *script* começa testando se a quantidade dos parâmetros passados é zero `(0)`, quando então usaremos as duas linhas que citamos, para transformar os parâmetros passados via entrada primária (*stdin*) em parâmetros posicionais `($1, $2, ... $n)`. O `for` terminará quando alcançar a quantidade de parâmetros passados `($#)` e o comando `printf`, em seu primeiro parâmetro, `(%0${##}d)`, diz que a linha será formatada `(%)` com um decimal `(d)` com `${##}` algarismos, preenchido com zeros à esquerda `(0)`.

Veja isso para entender melhor: como já vimos na seção referente ao comando `expr`, `${#var}` devolve o tamanho da variável `var` e também já vimos em passagem de parâmetros que `$#` retorna a quantidade de parâmetros passados. Juntando os dois, vemos que `${##}` devolve quantos algarismos tem a quantidade de parâmetros.

Uma vez montada a lista, é só passá-la para o comando `tr`, cuja opção `-T` diz que a formatação não é para impressora e o `-8` é para dividir a saída em 8 colunas

(trabalhar para que? O *Shell* já tem comando pronto para tudo!)

==========

Na página 298, antes do parágrafo Conforme você pode ... inserir:

Já tinha mandado a 10ª edição deste livro para a editora quando o Alfredo Casanova, colega da lista *shell-script* do *yahoo groups*, mandou a seguinte dica para a nossa lista:

"Só compartilhando uma funçãozinha que fiz aqui pra desenhar caixas de mensagem (só funciona para mensagens com uma linha, se alguém quiser alterar, fique à vontade)"

E nos brindou com esse código:

```
function DrawBox
{
    string="$*";
    tamanho=${#string}
    tput setf 3; printf "\e(0\x6c\e(B"
    for i in $(seq $tamanho)
        do printf "\e(0\x71\e(B"
    done
    printf "\e(0\x6b\e(B\n"; tput sgr0;
    tput setf 3; printf "\e(0\x78\e(B"
    tput setf 4; tput bold; echo -n $string; tput sgr0
    tput setf 3; printf "\e(0\x78\e(B\n"; tput sgr0;
    tput setf 3; printf "\e(0\x6d\e(B"
    for i in $(seq $tamanho)
        do printf "\e(0\x71\e(B"
    done
    printf "\e(0\x6a\e(B\n"; tput sgr0;
}
```

Seu uso seria da seguinte forma:

```
$ DrawBox Qualquer frase que caiba no terminal
```

```
Qualquer frase que caiba no terminal
```

Só que essa caixa é azul `(tput setf 3)` e as letras são vermelhas, com ênfase `(tput setf 4; tput bold)`.

Mas observe a tabela a seguir, ela explica os códigos gerados por esse monte de `printf` estranho:

O printf	Produz
\e(0\x6c\e(B	┌
\e(0\x71\e(B	─
\e(0\x6b\e(B	┐

O printf	Produz
\e(0\x78\e(B	│
\e(0\x6d\e(B	└
\e(0\x6a\e(B	┘

Como eu tinha acabado de escrever os exemplos que vimos acima e ainda estava com eles na cabeça, sugeri que o `for` que ele usou para fazer as linhas horizontais fosse trocado, assim substituiríamos:

```
for i in $(seq $tamanho)
    do printf "\e(0\x71\e(B"
done
```

Por:

```
printf -v linha "%${tamanho}s" ' '
printf -v traco "\e(0\x71\e(B"
echo -n ${linha// /$traco}
```

Tudo *Shell* puro, pois o `for`, `printf` e o `echo` são *bultins*, mas como o `printf` dentro do `for` seria executado tantas vezes quantos traços horizontais houvessem, imaginei que a minha solução fosse um pouco mais veloz e pedi-lhe para testar os tempos de execução, não sem antes apostar um chope. Ele criou 2 *scripts* um que executava mil vezes a função que ele havia proposto e outro que fazia o mesmo com a minha solução. Não vou dizer qual foi a forma mais veloz, porém lhes adianto que o Alfredo está me devendo um chope... ;)

O código otimizado ficaria assim:

```
function DrawBox
{
    string="$*";
    tamanho=${#string}
    tput setf 3; printf "\e(0\x6c\e(B"
    printf -v linha "%${tamanho}s" ' '
    printf -v traco "\e(0\x71\e(B"
    echo -n ${linha// /$traco}
    printf "\e(0\x6b\e(B\n"; tput sgr0;
    tput setf 3; printf "\e(0\x78\e(B"
    tput setf 4; tput bold; echo -n $string; tput sgr0
    tput setf 3; printf "\e(0\x78\e(B\n"; tput sgr0;
    tput setf 3; printf "\e(0\x6d\e(B"
    printf -v linha "%${tamanho}s" ' '
    printf -v traco "\e(0\x71\e(B"
    echo -n ${linha// /$traco}
    printf "\e(0\x6a\e(B\n"; tput sgr0;
}
```

Então agora, voltando aos nossos exemplos de passar uma linha por todo o terminal, posso sugerir uma outra (e mais bonita) solução:

- ✔ Para passar uma linha por todo o terminal (Dica 3)

```
printf -v linha "%$(tput cols)s" ' '
printf -v traco "\e(0\x71\e(B"
echo ${linha// /$traco}
```

==========

Na página 321, trocar o exemplo que começa em `$var=x` e termina com Variável ficou com o valor: por:

```
$ Var1=10; unset Var2; echo Var1 tem 10${Var2:+ e Var2 tem $Var2}
Var1 tem 10
$ Var1=10; Var2=; echo Var1 tem 10${Var2:+ e Var2 tem $Var2}
Var1 tem 10
$ Var1=10; Var2=20; echo Var1 tem 10${Var2:+ e Var2 tem $Var2}
Var1 tem 10 e Var2 tem 20
```

Como você pôde ver, enquanto inexistiu `$Var2` ou seu valor foi nulo, não houve a expansão do parâmetro. Isso aconteceu somente após a variável ser valorada.

==========

Trocar o texto que começa na pág 327

- ${parâmetro}

até a seguinte linha de comando da pág 328:

```
Separados_Por_Sublinha
```

Por:

- ${parâmetro?MSG ERRO}

Caso `parâmetro` não seja uma variável definida, `MSG ERRO` será exibida na tela e o programa terminará passando código de retorno `($?)` igual a `1`.

```
$ : ${PATH?Impossível continuar, \
    caminho dos arquivos não declarado}     Não dá saída
$ unset PATH
$ : ${PATH?Impossível continuar, \
   caminho dos arquivos não declarado}
bash: PATH: Impossível continuar, caminho dos arquivos não declarado
```

Supondo que essa fosse uma consistência feita no início do programa e a intenção era somente ver se a variável estava definida ou não, sem dar nenhuma saída para a tela, a não ser no caso da variável não estar definida, foi usado a expansão de parâmetros como entrada do comando `:`, que não faz nada, obtendo desta forma somente uma mensagem em caso de erro.

==========

No capítulo 7, antes do título: Ganhando o jogo mais coringas (desculpe mas estava sem o livro para ver a página deste título), incluir:

Um colega postou a sua dúvida na lista de *Shell* do *Yahoo*:

Tenho três variáveis, `$var1`, `$var2` e `$var3`. Quero gerar todos os valores possíveis no formato `$var1/$var2/$var3`, sendo que:

`$var1` varia de `1` a `14`;

`$var2` varia de `1` a `8`;

`$var3` varia de `1` a `64`.

Fui o primeiro a responder e logo de cara sugeri o mais óbvio, isto é, três comandos `for` encadeados, fazendo assim:

```
for ((var1=1;var1<=14; var1++))
{
    echo $var1/$var2/$var3
    for ((var2=1;var2<=8;var2++))
    {
        echo $var1/$var2/$var3
        for ((var3=1; var3<64; var3++))
        {
            echo $var1/$var2/$var3
        }
    }
}
```

Logo depois, o Itamar Santos de Souza manda essa preciosidade:

```
for termo in $(echo {1..14}/{1..8}/{1..64})
do
    echo "$termo"
done
```

Matou a pau! Dessa forma, ele não usou nenhuma variável e fez somente um `for`, onde a execução é feita de trás para a frente, isto é, a cada vez que o 3º índice chega a `64`, o 2º é incrementado em `1` e este, por sua vez, quando chega a `8`, incrementa o 1º e o *loop* terminará a execução quando este índice chegar a `14`.

Mas roubando a sua ideia de usar expansão de chaves, ainda consegui gerar um one-liner, cujo tempo de execução é o menor dentre as 3 opções citadas. Veja:

```
echo {1..14}/{1..8}/{1..64} | tr ' ' '\n'
```

Pessoal, coloquei estes exemplos para enfatizar o que eu sempre digo: "nunca pergunte se uma determinada tarefa pode ser implementada em *Shell*. A pergunta correta é: qual é a melhor forma de se implementar esta tarefa em *Shell*".

==========

No final da página 336, incluir:

Como já vimos, no Bash, para atribuirmos valores a um vetor, basta fazer a atribuição colocando os valores entre parênteses:

```
Praias=(Copacabana Ipanema Leblon)
```

Vamos ver o que isso gerou:

```
for ((i=0; i<=5; i++))
do
    printf "%d%20s\n" $i "${Praias[i]}"
done
0           Copacabana
1              Ipanema
2               Leblon
3
4
5
```

Gostaria de incluir outras praias, aí eu tento fazer:

```
Praias=([3]="Arraial do Cabo" [4]="Búzios" [5]="Cabo Frio")
```

Vamos aproveitar o `for` anterior e ver como ficou:

```
for ((i=0; i<=5; i++))
do
    printf "%d%20s\n" $i "${Praias[i]}"
done
0
1
2
3     Arraial do Cabo
4              Búzios
5           Cabo Frio
```

Epa, dessa forma, escrevi por cima, isto é, substituí o conteúdo existente por um novo valor. Vamos restaurar os valores iniciais das praias:

```
Praias=(Copacabana Ipanema Leblon)
```

Para anexar outros valores a esse vetor, temos de trocar atribuição pelo modelo de pré incrementação típico da aritmética no *Shell* `(+=)`, veja:

```
Praias+=("Arraial do Cabo" Búzios "Cabo Frio")
```

Mais uma vez aproveitando o `for` anterior, vem:

```
for ((i=0; i<=5; i++))
do
    printf "%d%20s\n" $i "${Praias[i]}"
done
0          Copacabana
1             Ipanema
2              Leblon
3     Arraial do Cabo
4              Búzios
5           Cabo Frio
```

DICA: Você também pode utilizar o mesmo artifício (referindo-me ao `+=`) como um operador de concatenação de cadeias.

Veja:

```
Lista=""
for Letra in {A..F}
do
    Lista+=$Letra"-"
done
echo $Lista
A-B-C-D-E-F-
```

Ou seja, usando o operador `+=` nós concatenamos o valor da variável `Lista` com cada uma das letras geradas pela expansão de chaves `({A..F})`, seguida de um traço `(-)`.

No final da página 338 trocar a frase: Existe uma outra forma de fazer o mesmo. por:

O que veremos nessa seção é quase tudo aplicação das expansões de parâmetros que vimos na seção Construções com parâmetros e variáveis, se você não se recorda, é melhor voltar lá e dar uma olhadinha.

Existe uma outra forma de fazer o mesmo que fizemos com o vetor Frutas, isto é, listar todos os seus elementos. E nesse mesmo parágrafo, emendar o resto: Para mostrá-la...

Na página 341, excluir de: Experimente agora... até a última linha do exemplo, começada por Alcançar o céu ... e substituir por:

Alguns usos de expansão de parâmetros usados com vetores. O vetor referente a nossas ensolaradas `Praias` está assim:

```
$ echo "${Praias[@]}"
Copacabana Ipanema Leblon Arraial do Cabo Búzios Cabo Frio
```

Para tirar os `Cabo:`

```
$ echo ${Praias[@]/Cabo/}
Copacabana Ipanema Leblon Arraial do Búzios Frio
```

Para trocar os `C` e `c` por `N:`

```
$ echo "${Praias[@]//[Cc]/N}"
NopaNabana Ipanema Leblon Arraial do Nabo Búzios Nabo Frio
```

Para tirar à direita tudo após cada `C:`

```
$ echo "${Praias[@]%C*}"
 Ipanema Leblon Arraial do  Búzios
```

Para tirar tudo à esquerda antes de cada primeiro `o:`

```
$ echo "${Praias[@]#*o}"
pacabana Ipanema n  Cabo s  Frio
```

Para tirar tudo à esquerda antes de cada último `o:`

```
$ echo "${Praias[@]##*o}"
pacabana Ipanema n  s
```

Agora não mais se trata de usar expansão de parâmetros com vetores, mas como a manipulação de vetores, como já vimos, nos permite fazer operações aritiméticas com seus índices, poderíamos resgatar `${Praias[4]}` de diversas formas:

```
$ echo ${Praias[3+1]}
Búzios
$ echo ${Praias[2*2]}
Búzios
```

O que ainda não tinhamos visto é que esta aritimética aceita números relativos, assim sendo para resgatar este mesmo campo, que é o penúltimo do vetor, podemos fazer:

```
$ echo ${Praias[-2]}
Búzios
```

Veja mais:

```
$ echo ${Praias[-1]}
Cabo Frio
```

Deve ter dado para notar que índices negativos são relativos ao fim do vetor assim `${Praias[-1]}` aponta para o último elemento do vetor e `${Praias[-2]}` para o

penúltimo.

Mais uma abordagem ligeiramente diferente, porém que ajuda a fazer *scripts* mais enxutos:

```
$ Motos=(honda yamaha harley\ davidson kawasaki agusta)
$ echo ${Motos[*]}
honda yamaha harley davidson kawasaki agusta
$ (IFS=$'\n'; echo "${Motos[*]}")   Parenteses para não alterar o meu IFS
honda
yamaha
harley davidson
kawasaki
agusta
```

Desta forma conseguimos listar cada elemento de um vetor em uma linha. O `$IFS` do *Shell* corrente não foi alterado, porque os parênteses mais externos, obrigam a criação de um novo *Shell* e neste sim, a variável foi alterada.

==========

Na pag. 359, tabela que enviei para a edição anterior era:

Sinal	Descrição
`EXIT`	Ativado ao término do *script*
`DEBUG`	Ativado ao fim de cada comando
`RETURN`	Ativado ao fim de função iniciada com comando `source`
`ERR`	Ativado sempre que um comando é mal sucedido

==========

Na página 360, antes da seção Funções, inserir:

Você também pode monitorar o fim de um programa `PRG` mandando um `kill -0 PRG`, pois o sinal zero `(0)` é neutro e por isso inócuo para o programa, mas caso `PRG` já tenha terminado, ocorrerá um erro que pode ser capturado por uma instrução de teste.

Veja isso:

```
$ sleep 30 &
[1] 21080
$ kill -0 $! && echo vivo || echo morto   $! é o PID do background
vivo
$ kill -0 $! && echo vivo || echo morto
bash: kill: (21080) - Processo inexistente
morto
```

Essa mensagem de erro poderia ser evitada, desviando-se a saída de erro padrão (*stderr*) para `/dev/null`, finalmente ficando assim:

```
$ sleep 30 &
[1] 21087
```

```
$ kill -0 $! 2> /dev/null && echo vivo || echo morto
vivo
$ kill -0 $! 2> /dev/null && echo vivo || echo morto
morto
[1]+  Concluído               sleep 30
```

==========

Na pág 364 antes do título Uma função "on error", inserir:

Mais duas funções simples e interessantes:

```
function Repete
{
#  Repete um caractere um determinado número de vezes
#+ Recebe:
#+ Tamanho final da cadeia
#+ e caractere a ser repetido
    local Var
    printf -v Var %$1s " "
    echo ${Var// /$2}
}

function EncheEsq
{
#  Preenche à esquerda com caractere especificado
#+ Recebe:
#+ Valor inicial da cadeia,
#+ Tamanho final e char de preenchimento
    local Var
    local Cadeia=${1// /^}          Trocando eventuais espaços pré existentes
    printf -v Var %$2s $Cadeia
    Var=${Var// /$3}
    echo "${Var//^/ }"              Restaurando espaços anteriores
}
```

Em ambas as funções fizemos um `printf` para dentro da variável `$Var`, preenchendo-a à esquerda com a quantidade de espaços em brancos necessários para completar o tamanho final. A expansão de parâmetros da linha seguinte, serve para trocar cada um dos espaços em branco pelo caractere de preenchimento especificado.

Agora é importante realçar na função `EncheEsq` que quando criamos a variável `$Cadeia`, trocamos todos os espaços em branco pré existentes por um circunflexo `(^)`, para que esses espaços não fossem trocados pelo caractere de preenchimento também.

```
$ Repete 10 -
----------
$ Repete 15 .
...............
$ EncheEsq abc 10 -
-------abc
$ EncheEsq 123,45 10 \#
####123,45
```

Provavelmente você já notou que se usar a função `EncheEsq`, usando como valor inicial da cadeia um campo vazio, terá como saída o mesmo resultado que a função

`Repete` geraria. Veja:

```
$ EncheEsq "" 5 -
-----
$ Repete 5 -
-----
```

Se você precisar de preencher à direita, basta inverter o sinal do `printf.` Assim poderíamos ter uma função `EncheDir,` assim:

```
function EncheDir
{
#  Preenche à direita com caractere especificado
#+ Recebe:
#+ Valor inicial da cadeia,
#+ Tamanho final e char de preenchimento
    local Var
    local Cadeia=${1// /^}          Trocando eventuais espaços pré existentes
    printf -v Var %-$2s $Cadeia
    Var=${Var// /$3}
    echo "${Var//^/ }"              Restaurando espaços anteriores
}
```

```
$ echo "Pague-se R$ 100,00 ($(EncheDir "Cem reais" 20 \#))"
Pague-se R$ 100,00 (Cem reais##########)
```

==========

Incluir na página 373 antes de O último exemplo para atestar a versatilidade...

Surgiu uma dúvida durante um treinamento que eu lecionava.

Um treinando perguntou:

Como posso listar as linhas incompletas de um arquivo, isto é, cada registro do meu arquivo deveria ter 3 campos separados por um dois pontos `(:)` mas alguns estão incompletos. Como faço para listá-los?

O arquivo era assim:

```
$ cat notas
Aluno:Nota 1:Nota 2
Juliana Duarte Neves:9,0:9,5
Paula Duarte Neves:9,5:9,5
Silvina Duarte:9,0
Flamary Coutinho:6,0
Rosana Coutinho:6,5:7,2
```

A minha proposta foi fazer o seguinte:

```
$ grep -Eo '^[^:]+:[0-9,]+$' notas
Silvina Duarte:9,0
Flamary Coutinho:6,0
```

Ou seja, a partir do início `(^),` para casar o nome, procurei tudo que não fosse dois pontos `([^:]+),` seguido de dois pontos `(:),` por sua vez seguido de um ou mais algarismos e vírgula `[0-9,]+` e se encerrando aí `($).`

Ele falou: está quase bom, mas são muitos alunos e gostaria que já viesse o total de registros incompletos. Então eu fiz:

```
$ grep -Eo '^[^:]+:[0-9,]+$' notas |
    (sleep 0.4; tee >(wc -l))
Silvina Duarte:9,0
Flamary Coutinho:6,0
2
```

O `sleep` foi colocado para que desse tempo da resposta do `wc -l` aparecesse antes de encerrar o comando.

E se quisesse esnobar as qualidades do *Shell*, eu poderia editar `(sed)` a última linha da saída `($)`, substituindo `(s)` seu início `(^)`, pelo texto de totalização, veja:

```
$ grep -Eo '^[^:]+:[0-9,]+$' notas |
    (sleep 0.4; tee >(wc -l))       |
    sed '$ s/^/Qtd. de registros incompletos: /'
Silvina Duarte:9,0
Flamary Coutinho:6,0
Qtd. de registros incompletos: 2
```

==========

Na pág 379 trocar o título para Brincando com o Nautilus e o Caja e em seguida inclua:

Quando escrevi esse livro, o navegador padrão do gnome era o Nautilus hoje, toda hora vemos pessoas utilizando o Caja (que ainda não consegui enxergar nenhuma dirença para o primeiro), mas a verdade é que em termos de script o caja faz tudo que o Nautilus.

Como essa seção foi toda escrita para o Nautilus, deixo para vocês fazerem as correções necessárias a fim de não tornar a leitura redundante e enfadonha. As diferenças que podem afetar, serão mostradas.

Por exemplo, no parágrafo seguinte, está escrito: "O Nautilus permite que você crie seus próprios *scripts* e ..." e assim continuará, mas se você usa o Caja, leia "O **Caja** permite que você crie seus próprios *scripts* e ...".

==========

Na pág. 380 troque a última frase do 1º. Parágrafo, por::

No Nautilus, seu caminho completo é $Home/gnome2/nautilus-scripts, já no Caja o caminho é $HOME/.config/caja/scripts/

==========

Pág 381, trocar o 1º. parágrafo por: A tabela a seguir mostra as variáveis que o Nautilus passa, para serem usadas em *scripts* (as do Caja veremos logo após)

==========

Na pag 382, antes de: As definições anteriores... Incluir:

A tabela a seguir mostra as variáveis que o Caja passa, para serem usadas em *scripts*

Nome da variável	Finalidade
`CAJA_SCRIPT_SELECTED_FILE_PATHS`	Caminhos delimitados por fim-de-linha para os arquivos selecionados (apenas se locais)

Nome da variável	Finalidade
`CAJA_SCRIPT_SELECTED_URIS`	URIs delimitadas por fim-de-linha para os arquivos selecionados
`CAJA_SCRIPT_CURRENT_URI`	URI para a localização atual
`CAJA_SCRIPT_WINDOW_GEOMETRY`	Posição e tamanho da janela atual
`CAJA_SCRIPT_NEXT_PANE_SELECTED_FILE_PATHS`	Caminhos delimitados por fim-de-linha para os arquivos selecionados no painel inativo de uma janela dividida (apenas se locais)
`CAJA_SCRIPT_NEXT_PANE_SELECTED_URIS`	URIs delimitadas por fim-de-linha para os arquivos selecionados no painel inativo de uma janela dividida
`CAJA_SCRIPT_NEXT_PANE_CURRENT_URI`	URI para a localização atual no painel inativo de uma janela dividida

==========

Na página 400, antes do título "Macetes, macetes & macetes", incluir a Seção a seguir:

Copiar e colar nas áreas de transferência

Quando mandei para a editora o material para a 10ª edição desse livro, fui convencido que seria melhor para o leitor se os arquivos que usei e criei ao longo desse trabalho, fossem disponibilizados em um site e não mais em um CD, como ocorreu até a 9ª edição.

Eu tinha quase 200 arquivos que teriam de ser copiados um a um para uma página, isto é, `cat arquivo,` depois um `<CTRL>+<SHIFT>+C` para copiar de uma sessão X (*Gnome*) ir ao *site* para escrever o nome do arquivo em negrito `<B>arquivo></B>` e fazer um `<CTRL>+V` para colar seu conteúdo e deixar uma linha em branco para separá-lo do próximo. Isso daria uma trabalheira dos diabos, mesmo fazendo pelo *Nautilus* (gerenciador de arquivos que como mostramos aceita o uso de *scripts*).

Como o *Shell* vicia pois as soluções são sempre pequenas e imediatas e como eu já tinha tomado conhecimento de um utilitário para manipular a área de transferência (*clipboard*), dei uma fuçada rápida na internet e descobri não um, mas dois: o `xclip` e o `xsel.` Como a finalidade dos dois é a mesma, resolvi escrever somente sobre o `xclip` que é o mais usado e que já vem instalado por padrão no *Debian*, apesar de saber que o `xsel` tem um pouco mais de funções.

A área de transferência: você sabia?

Sempre imaginei e acho que muitos de vocês também que a área de transferência fosse um *buffer* que recebia o dado copiado, mas a coisa é bem mais interessante. Você já deve ter notado que se arrastar o *mouse* em um texto e clicar o botão do

centro (ou esquerdo e direito juntos), esse texto será copiado para o local onde está o cursor (se você não conhecia isso, experimente. Agiliza muito o copia->cola) e se você fizer um **<CTRL>+V,** será listado algo que você guardou com um **<CTRL>+C** e não com a arrastada do *mouse*. Pois é isso que define dois dos *buffers* que existem, a saber:

- **XA_PRIMARY** - Este é o padrão (*default*) é ele que recebe os dados quando você simplesmente arrasta o *mouse* sobre um texto. Esse texto pode ser recuperado clicando no botão do meio ou no da esquerda e da direita simultaneamente;

  Um exemplo que uso para dar água na boca dos meus amigos que usam rwin. No terminal eu faço:

  ```
  $ uptime | xclip
  ```
  uptime devolve o tempo sem dar boot

  E aqui nesse texto vou clicar na linha a seguir com o botão do meio, veja:

  ```
  00:03:54 up 50 days, 13:37, 4 users, load average: 0,39, 0,28, 0,37
  ```

  Já pude dar esse exemplo, porque como já havia dito o primário é o valor padrão e portanto mais simples

- **XA_CLIPBOARD** - Este recebe os dados com **<CTRL>+C** e recupera-os com **<CTRL>+V.**

  É interessante notar que quando se copia para o primário somente este receberá a cópia, ao passo que se a cópia for feita para o *clipboard*, ambos receberão essa cópia.

- **XA_SECONDARY –** Tentei muito colocar dados nesse *buffer*, mas não consequi nem com uso dos botões do *mouse*, nem com os macetes/atalhos do teclado e, para piorar, o tio Google não me apresentou nada sobre o uso desse cara. Mas depois de suar os bigodes, descobri que consigo municiá-lo via programas (**xclip** e **xcel**), o que me permite ter uma terceira área de transferência disponível.

Opções do comando

Este comando tem inúmeras opções, mas como sempre, mostrarei somente as mais usadas. São elas:

Opção	Ação
-i	O **xclip** receberá dados da entrada primária ou arquivo(s)
-o	O **xclip** enviará dados para a saída
-selection	Seleciona *buffers*. Opções: **primary; secondary** ou **clipboard**[1]
-f	Não filtra a saída. Joga-a na tela
-t	Seleciona tipo de alvo que será copiado

[1] - Para definir a seleção basta usar o primeiro caractere de cada um dos *buffers*. Assim basta usar **p, s** ou **c** respectivamente. Você também pode abreviar

`-selection` escrevendo somente `-sel.`

Bem, basicamente já vimos toda a teoria, vamos aos exemplos para entendê-la.

Nos exemplos a seguir, as transferências foram efetuadas entre terminal, `gedit` e arquivo.

O mais simples. Transferindo o conteúdo de `arquivo` para o *buffer* primário:

```
$ xclip -i -selection p arquivo
```

Mas como a opção `-i` é padrão e `primary (p)` é o padrão da opção `-selection,` esse exemplo poderia ser escrito assim:

```
$ xclip arquivo
```

Pronto! O conteúdo de arquivo já está armazenado no *buffer* primário. Você verá em diversos lugares (diria até que a maioria) isso sendo feito dessa maneira:

```
$ cat arquivo | xclip
```

Isso é uma perda de tempo e é uma forma de escrever mal o *Shell*.

Agora que você já copiou o arquivo, estando no terminal você pode devolvê-lo de duas formas:

1. Clicando no botão do meio ou no da esquerda e da direita simultaneamente (em qualquer eventualidade que você precise do botão do meio e não o tenha (e isto é frequente em *notebooks*) ele sempre pode ser substituído por um clique simultâneo nos dois botões);
2. Executando o comando:

   ```
   $ xclip -o
   ```

   Onde o conteúdo do *buffer* será passado para a saída primária (opção `-o`)

A diferença das duas formas é que na primeira, os dados são interpretados e na segunda não. Caso `arquivo` fosse um *script*, da primeira forma ele seria executado, da segunda seria somente listado.

Você pode redirecionar a saída do `xclip` para um comando, de forma a selecionar o texto que você deseja. Para te mostrar isso, vou copiar um texto que escrevi no `gedit` e vou baixá-lo no terminal, precedendo todos os números `n` por um jogo da velha, resultando `#n.` Escreva a linha a seguir no `gedit` (ou no `writer`)

```
1 2 3 de Oliveira 4
```

Arraste o *mouse* por cima, dê um `<CTRL>+C` e no terminal faça:

```
$ xclip -o -sel c | sed -r 's/([0-9]+)/#\1/g'
#1 #2 #3 de Oliveira #4
```

No `gedit,` quando copiei com `<CTRL>+C,` o texto copiado foi para o *clipboard* daí termos dito que a saída `(-o)` viria deste *buffer* `(-sel c,` que é a forma abreviada de `-selection clipboard).` Quanto ao `sed,` na entrada os parênteses montam grupos com todas a sequências numéricas `[0-9]` com um ou mais caracteres `(+)` e na saída as sequências lidas são substituídas por elas mesmas `(\1)` precedida por um jogo da velha `(#).` Se no final do comando não tivéssemos colocado um `g` (*global*), somente a primeira sequência numérica seria modificada.

Só para reforçar a ideia do `sed,` veja como colocar todos os números entre arrobas `(@).`

```
$ sed -r 's/([0-9]+)/@\1@/g' <<< '1: um, 20: vinte, 300: trezentos'
@1@: um, @20@: vinte, @300@: trezentos
```

Lá no início dessa seção, quando apresentava as áreas de transferência, eu disse que quando você copiava um determinado texto para o *clipboard*, o conteúdo do *buffer* primário também receberia esse texto. Agora cabe complementar: a não ser que a cópis seja feito por um

Como os dados do `xclip` podem vir via *pipe* ou via nome do arquivo a ser copiado, para não ter trabalho, desenvolvi uma função que coleta os dados da entrada primária ou de um arquivo e os coloco no *clipboard*. Todo programador deve ter um arquivo no qual ele tenha as funções que mais usa. Essa função é assim:

```
function LeEntrada
{
if ! [[ -t 0 ]]  #  Testa se file descriptor 0 (entrada
                 #+ primária) está aberto no terminal
then
    echo -n "$(< /dev/stdin)" |
        xclip -selection c && \
            echo "Copiado para clipboard"
else
    if [[ -z "$@" ]]  # Cadê o(s) parâmetro(s)
    then
        echo "Uso:
        $0 ARQ - Manda arquivo ARQ p/ clipboard
        CMD | $0 - Manda saída de CMD p/ clipboard"
        return 1
    fi
    # Então parâmetro passado foi um arquivo.
    if [[ ! -f "$@" ]]
    then
        echo  Arquivo $@ não existe
        return 1
    else
        xclip -i -selection clipboard "$@"
        echo "Arquivo "$@" copiado para clipboard"
    fi
fi
}
```

Digamos que você queira mandar uma saída simultaneamente para o *buffer* primário e para a área de transferência (*clipboard*). Se você fizer:

```
echo "Olá" | xclip -i -sel c | xclip -i -sel p
```

Isso não funcionará porque o primeiro `xclip` não gerará saída alguma para o segundo. Nesse caso é necessário usar a opção `-f` que mandará o conteúdo do *buffer* para a saída. Veja:

```
$ echo "Olá" | xclip -i -sel c
$ echo "Olá" | xclip -i -sel c -f
Olá
```

Então, para mandar para ambos os *buffers*, o correto seria:

```
$ echo "Olá" | xclip -i -sel c -f | xclip -i -sel p
```

Para encerrarmos esse assunto, só mais um exemplinho bobo (como quase todos, dos poucos, que existem no `man`)

```
$ xclip -t text/html index.html
```

Onde a opção `-t` especifica que o alvo será um arquivo de `html`.

Só para não dizer que não falei sobre o *buffer* secundário, veja esse exemplo, onde uso as três áreas de transferências:

```
$ echo Primario   | xclip -sel p
$ echo Secundário | xclip -sel s
$ echo Clipboard  | xclip -sel c
$ xclip -o -sel p
Primario
$ xclip -o -sel s
Secundário
$ xclip -o -sel c
Clipboard
```

==========

Na pág 417, na tabela, trocar `FMR` por `FNR` e inserir a seguinte linha:

`IGNORECASE` Com o valor zero, seu padrão, leva em conta a caixa em comparações

==========

Na tabela da pag 424 tirar o x que está entre parênteses ficando: `rand ()`. Inserir uma linha após essa na tabela, com o seguinte conteúdo:

`srand([x])` inicializa com `x` a semente geradora de randômicos

==========

E logo após a tabela colocar:

`rand ()` gerará randômicos a partir do mesmo número ou semente, a cada vez que seu programa solicitar um randômico então é conveniente que você gere uma nova semente para cada randômico, e isso se faz com `srand()`.

Se o argumento `x` de `srand ()` for omitido, a data e hora corrente serão usadas como semente. Costumo fazer:

```
$ awk 'BEGIN { srand(systime() + PROCINFO["pid"])
    print rand() }'
0.357732
```

==========

Na tabela da pag 425, alterar a linha:

`length (c1)` Retorna o tamanho da cadeia `c1` ---

por:

`length (c1|v1)` Retorna o tamanho da cadeia `c1` ou a quantidade de elementos do vetor `v1` ---

Na mesma tabela, incluir as seguintes linhas:

`asort(V1 [, V2 [, MODO]])`	Classifica vetor `V1` com saída em `V2` (se declarado). `MODO` é a direção (`ascending` ou `descending`) e/ou critério (`number` ou `string`) ---

asorti(V1 [, V2 [, MODO]])	Idêntica a **asort,** porém classifica pelos índices ---