Curso de Pós-Graduação
“Lato Sensu” (Especialização) a Distância
Administração em Redes Linux

AUTOMAÇÃO DE TAREFAS

Herlon Ayres Camargo

Universidade Federal de Lavras – UFLA
Fundação de Apoio ao Ensino, Pesquisa e Extensão – FAEPE
Lavras – MG

PARCERIA

UFLA – Universidade Federal de Lavras

FAEPE – Fundação de Apoio ao Ensino, Pesquisa e Extensão

REITOR

Antônio Nazareno Guimarães Mendes

VICE-REITOR

Ricardo Pereira Reis

DIRETOR DA EDITORA

Marco Antônio Rezende Alvarenga

PRÓ-REITOR DE PÓS-GRADUAÇÃO

Joel Augusto Muniz

PRÓ-REITOR ADJUNTO DE PÓS-GRADUAÇÃO "LATO SENSU"

Marcelo Silva de Oliveira

COORDENADOR DO CURSO

Kátia Cilene Amaral Uchôa

PRESIDENTE DO CONSELHO DELIBERATIVO DA FAEPE

Edson Ampélio Pozza

EDITORAÇÃO

Grupo Ginux (`http://ginux.comp.ufla.br/`)

IMPRESSÃO

Gráfica Universitária/UFLA

Ficha Catalográfica preparada pela Divisão de Processos Técnicos da Biblioteca Central da UFLA

Camargo, Herlon Ayres

Automação de Tarefas / Herlon Ayres Camargo. -- Lavras: UFLA/FAEPE, 2005.

152 p. : il. - Curso de Pós-Graduação "Lato Sensu" (Especialização) a Distância: Administração em Redes Linux.

Bibliografia.

1. Informática. 2. Linux. 3. Scripts. 4. Bash. 5. Perl. 6. Rpm. 7. Cron I. Camargo, H.A. II. Universidade Federal de Lavras. III. Fundação de Apoio ao Ensino, Pesquisa e Extensão. IV. Automação de Tarefas.

CDD-005.43

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1

INTRODUÇÃO

A rotina de trabalho de um administrador de sistemas e/ou de redes, além de exigir muita responsabilidade, pode exigir também muito esforço para realizar as várias tarefas de sua função. Entre as principais tarefas, pode-se destacar a realização de *backups* periódicos de diversos tipos de arquivos, gerenciamento de contas de usuário, gerenciamento de serviços de rede, policiamento da política de uso e de segurança da rede, instalação e atualização de pacotes, entre tantas outras. Geralmente são tarefas repetitivas, metódicas e que exigem muita atenção. Os sistemas UNIX e seus derivados possuem recursos nativos para que essas tarefas sejam automatizadas, reduzindo-se assim, o trabalho do administrador, a chance de ocorrerem erros e esquecimentos, além de tornarem a execução das tarefas mais rápida.

O objetivo deste texto é ensinar ao leitor o uso básico de ferramentas poderosas para automatização de tarefas. Ao final, o leitor será capaz de criar *scripts* e procedimentos para automatizar tarefas. Não é objetivo deste texto ensinar o leitor a configurar sistemas e redes, nem lógica de programação. O autor pressupõe que o leitor tenha conhecimento de programação, noções básicas da sintaxe da linguagem de programação C e conhecimento mediano dos principais comandos usados em Linux.

Este texto encontra-se organizado da forma como se segue. O Capítulo 2 apresenta as características do *shell* para programação de *scripts*. São apresentados os conceitos de *shell* e *shell-script*, além de citar os principais *shells* encontrados. O capítulo se concentra na sintaxe de programação utilizada por *shell-scripts*, de forma que o usuário poderá adaptar comandos utilizados no *prompt* para serem executados de forma sistemática em *scripts*. O capítulo encerra com a primeira parte de um exemplo prático que abrange a maioria dos conceitos estudados.

O Capítulo 3 aborda as ferramentas mais utilizadas para manipulação de textos, no ambiente de administração de sistemas e/ou redes. Essas ferramentas possibilitam a extração rápida de informações em arquivos de *log*, alteração e correção de textos em vários arquivos ao mesmo tempo, entre outra facilidades. É abordado o editor de textos orientado por fluxo Sed, a linguagem de programação voltada para bancos de dados textuais Awk, e o uso de Expressões Regulares para especificação de padrões. Esse capítulo encerra com

a segunda parte de um exemplo prático, iniciado no Capítulo 2, abrangendo conceitos dos Capítulos 2 e 3.

No Capítulo 4 é mostrada a linguagem de programação Perl, que também é muito utilizada na administração de sistemas e redes. É uma linguagem interpretada com alto nível de abstração e portabilidade, e possui muitas facilidades para trabalhar com processamento de textos. O capítulo apresenta os tipos de variáveis utilizadas, mostra a sintaxe de programação, Expressões Regulares no contexto de Perl e termina com um exemplo prático abrangendo a maioria dos recursos estudados.

O Capítulo 5 é de autoria de Bruno de Oliveira Schneider, professor do DCC da UFLA, e foi retirado na íntegra de [Schneider (2003)], com a sua devida permissão. O capítulo trata do uso de ferramentas de desenvolvimento que facilitam a configuração, compilação e instalação de programas a partir de seu código-fonte. É abordado o uso do programa *make* e das *autotools*.

Também de autoria de Bruno de Oliveira Schneider, o texto do Capítulo 6 foi retirado na íntegra de [Schneider (2003)], com sua devida permissão. O capítulo aborda o uso do Gerenciador de Pacotes *Red Hat* (RPM). O RPM é capaz de construir um pacote com arquivos binários, de configuração, de dados e de documentação que formam um aplicativo. Ele também é capaz de instalar, desinstalar e atualizar pacotes no formato RPM, que é o nome dado aos pacotes que estão no formato desse gerenciador, de forma simples e rápida.

O agendamento de tarefas é discutido no Capítulo 7, onde são mostrados os programas agendadores Cron e At. O capítulo explica a sintaxe usada nos arquivos de configuração desses programas. Com eles é possível definir uma data e horário para uma tarefa ser executada, sem a necessidade da presença do administrador de sistemas e/ou de redes. Essas tarefas podem ser desde um *backup* periódico até uma atualização do sistema em um horário de pouco uso.

Herlon Ayres Camargo, autor do texto, natural de Barbacena/MG, é Engenheiro Eletricista pela Universidade Federal de São João Del-Rei/MG (UFSJ), mestre em Engenharia Elétrica pelo Centro de Pesquisa e Desenvolvimento em Engenharia Elétrica (CPDEE) da Universidade Federal de Minas Gerais (UFMG). Foi professor do Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG) de 1997 a 2002, e atualmente é professor do Curso Superior de Tecnologia em Desenvolvimento de Sistemas Distribuídos e do Curso Técnico em Informática da Escola Agrotécnica Federal de Barbacena/MG, desde 2003, onde também desenvolve as atividades de administrador de redes e de suporte em *Software* Livre. Trabalha com informática desde 1991 e conheceu o Linux em setembro de 2000, através da distribuição Conectiva Linux 5, onde se apaixonou pelo sistema e pela filosofia de liberdade em *software*, fazendo com que abandonasse, definitivamente, seu diploma de Engenheiro Eletricista para trabalhar exclusivamente com Linux e Redes de Computadores.

O autor quer deixar claro os agradecimentos ao Professor Bruno de Oliveira Schneider, pela gentileza de ter “emprestado” os conteúdos dos Capítulos 5 e 6, para que pudessem completar o objetivo do texto, que é a automação de tarefas em sistemas Linux.

2

SHELL-SCRIPT

2.1 INTRODUÇÃO

Este capítulo mostra como construir *scripts* para serem executados por um *shell*. Em primeiro lugar, são definidos os conceitos de *shell* e *shell-script*. Em seguida, são apresentados alguns dos principais *shells* usados e algumas de suas características para desenvolvimento de *scripts*. São apresentados comandos usados no Linux que são muito úteis em *shell-script*. O capítulo se concentra no desenvolvimento de *shell-scripts*, mostrando sua sintaxe e vários exemplos. No final do capítulo, é apresentado um exemplo prático e real, envolvendo a maioria dos recursos vistos neste capítulo.

Para mais informações sobre *shell-script*, o autor recomenda a leitura dos manuais dos comandos usados no *Bash*, e também [Cooper (2005)], [Neves (2003)] e [Michael (2003)]. Muita informação avulsa também pode ser encontrada em [Jargas (2004)].

2.1.1 *Shell*

Shell é um interpretador de linha de comandos que tem como finalidade principal aceitar os comandos digitados pelo usuário e traduzi-los para o *kernel*, atuando como uma interface primária entre o usuário e o *kernel*. O *shell* possui uma sintaxe própria de alto nível, que permite a escrita de *scripts* com estruturas semelhantes às de linguagens de programação, usando variáveis, estruturas condicionais e de repetição, leitura e escrita em arquivos, operadores e funções. A Figura 2.1 mostra uma janela de terminal com um *prompt* do *shell* esperando por comandos a serem digitados pelo usuário. Neste texto, os exemplos exibem um *prompt* de comandos conforme a Figura 2.1, ou seja, `[prompt]$`. Este *prompt* pode variar dependendo da configuração do seu sistema, podendo aparecer, nessa posição, outras informações.

2.1.2 *Shell-Script*

Um *shell-script* é um arquivo contendo vários comandos do *shell* e que são executados em seqüencia. Esse tipo de *script* é muito utilizado por administradores de sistemas e de redes, sendo uma das ferramentas mais poderosas para automatizar tarefas em Linux.

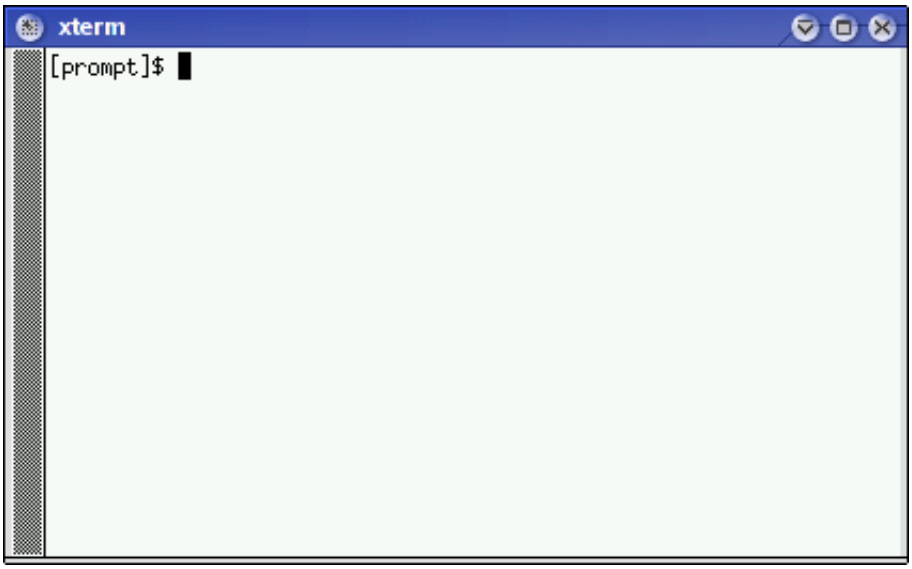

Figura 2.1: Janela de terminal exibindo o *shell*.

Permite total interação com o sistema, e pode ser executado através da linha de comandos ou chamado por outros *scripts*.

2.1.3 Principais *Shells*

Muitos *shells* foram desenvolvidos e alguns se sobressaíram e são muito utilizados. A seguir, são descritos os mais freqüentemente encontrados.

Bourne Shell

O *Bourne Shell* é o *shell* padrão do Unix e foi escrito por Stephen Bourne da *AT&T Bell Labs*. Chamado de "*Standard Shell*" ("*Shell* Padrão") é o *shell* mais utilizado porque muitos sistemas Unix-*Like* o trazem também. É representado por `sh`. Um detalhe é importante: é muito comum ser encontrado o arquivo `/bin/sh` no Linux, mas geralmente é um *link* simbólico para `/bin/bash`.

Bourne-Again Shell

O *Bourne-Again Shell* é o *shell* padrão do Linux, conhecido como "*Bash*", sendo um aperfeiçoamento do *Bourne Shell*. É representado por `bash`. Na maioria das distribuições Linux fica localizado em `/bin/bash`.

Korn Shell

O *Korn Shell* é também um aperfeiçoamento do *Bourne Shell*, tendo todos os seus comandos reconhecidos. Por ser uma evolução do *Bourne Shell*, está tendo grande aceitação nos sistemas Unix, sendo um forte candidato a substitui-lo. Foi desenvolvido por David Korn da *AT&T Bell Labs* e é representado por `ksh`.

C Shell

O *C Shell* é o mais utilizado nos ambientes *Berkeley* (BSD), tendo a sua sintaxe muito parecida com a da linguagem C. Foi desenvolvido por Bill Joy da *Berkeley University* e tem recursos superiores de programação de shell. É representado por `csh`.

Tenex C Shell

O *Tenex C Shell* é um aprimoramento do *C Shell* e muito utilizado em sistemas Linux, ficando atrás apenas do *Bash*. Possui a mesma sintaxe do *C Shell* com alguns comandos a mais, e é representado por `tcsh`.

2.2 CARACTERÍSTICAS DO *BASH* PARA *SHELL-SCRIPTS*

Como observado na seção anterior, o Bash é o *shell* padrão do Linux, sendo assim, as próximas seções apresentam algumas características do Bash em relação a elaboração de *shell-scripts*.

2.2.1 Metacaracteres

Metacaracter, ou caracter coringa, é um caracter que possui um significado especial, podendo representar um conjunto de caracteres. Os metacaracteres `*`, `?` e `[]`, quando encontrados, são substituídos por possíveis valores. A Tabela 2.1 resume o significado de cada metacaracter, trazendo também alguns exemplos.

2.2.2 Variáveis de Ambiente e de Sistema

As variáveis de ambiente e de sistema são variáveis definidas pelo próprio sistema e fazem parte do ambiente em que se está trabalhando. Algumas podem ser modificadas pelo próprio usuário, como por exemplo a variável `PATH`, mas a maioria é fixa. Elas podem ser utilizadas dentro de *shell-scripts* para se obter informações do sistema e do próprio usuário. A Tabela 2.2 mostra as principais e mais utilizadas variáveis de ambiente e de sistema.

Tabela 2.1: Metacaracteres usados no *Bash*.

Metacaracter	***Descrição***
*	Corresponde a qualquer *string* de zero ou mais caracteres, exceto um ponto (`.`) quando este é o primeiro caracter no nome de um arquivo. Exemplo: `var*` combina com `variavel`, `var01`, `varanda`, etc. O comando `ls *` combina com qualquer nome de arquivo exceto os que começam com ponto: combina com `relatorio`, `aviso.txt`, mas não combina com `.segredo`.
?	Corresponde a um único caracter qualquer. Exemplo: `var?` corresponde com `vara`, `var1`, `varb`, etc.
`[]`	Corresponde a um único caracter entre os presentes no interior dos colchetes. Um hífen pode definir um intervalo. Exemplo: `var[sgt]` combina com `vars`, `varg` e `vart`; `var[0-9]` combina com `var0`, `var1`, `var2`, ..., `var9`. Um sinal de exclamação (`!`) no interior do par de colchetes significa qualquer valor exceto os que estiverem entre os colchetes. Exemplo: `var[!abc]` só não combina com `vara`, `varb` e `varc`.

Tabela 2.2: Variáveis de ambiente e de sistema.

Variável	***Descrição***
`HOME`	Define o caminho completo do diretório *home* de um usuário.
`SHELL`	Identifica o *shell* do usuário e a sua localização.
`LOGNAME`	Identifica o nome de *login* do usuário.
`PATH`	Define o diretório em que o *shell* procura por comandos.
`TERM`	Define o tipo de terminal que o usuário está usando.
`HOSTNAME`	Identifica o nome da máquina.
`MAIL`	Define o endereço da caixa de correio do usuário.
`PWD`	Identifica o diretório corrente.

2.2.3 Redirecionamento de Entrada e Saída

Por default, *Bash* considera o teclado como entrada padrão, e o monitor de vídeo como saída padrão. Determinados comandos exigem entrada e saída de dados. Se nada

é especificado, esses comandos esperam que os dados sejam digitados no teclado e os resultados serão mostrados na tela.

Em alguns casos, é interessante gravar, num arquivo, os resultados gerados por um comando. Para isso, deve-se informar ao *Bash* que a saída de dados será gravada num arquivo. Outra situação seria a necessidade de fornecer uma grande quantidade de dados para um comando. Esses dados já poderiam estar gravados em um arquivo. Não é necessário que sejam digitados novamente no teclado, basta definir que a entrada do comando será feita através de um arquivo.

Pode acontecer também que um comando gere uma saída, e esta saída servirá de entrada para outro comando. Além disso tudo, se houver algum erro, pode-se gravar a mensagem de erro em um arquivo para uma análise posterior. O *Bash* é bastante flexível em relação a entradas e saídas de comandos e programas.

A Tabela 2.3 mostra os símbolos usados para fazer o redirecionamento de entrada e saída de dados. Na Figura 2.2 há alguns exemplos. Observe a saída `/dev/null`: tudo que for direcionado a ela irá "sumir", ou seja, não será gravado em arquivo algum, nem exibido na tela.

Tabela 2.3: Caracteres de redirecionamento de entrada e saída.

Caracter	***Descrição***
>	Redireciona a saída de um comando para um arquivo. Se o arquivo não existir, ele será criado. Se já existir, seu conteúdo será apagado e será gravado apenas a saída do comando.
>>	Redireciona a saída de um comando para um arquivo, incluindo-o no final do arquivo. Não apaga o conteúdo existente do arquivo. Se o arquivo não existir então será criado.
2>	Redireciona as mensagens de erro geradas por um comando para um arquivo. O arquivo será criado mesmo se não houver erro algum.
<	Redireciona a entrada de um comando de forma que ela venha de um arquivo em vez do terminal.
<<	Indica ao *shell* que o escopo de um comando começa na linha seguinte e termina quando encontrar uma linha cujo conteúdo seja o mesmo que vem após o sinal <<.
|	Conhecido como *pipe*, redireciona a saída de um comando para a entrada de outro.

```
[prompt]$ ls -l > conteudo.txt
[prompt]$ cat conteudo.txt
total 12
-rw-rw-r--  1 herlon herlon  0 2005-02-27 22:41 conteudo.txt
-rwxrwxr-x  1 herlon herlon 60 2005-02-27 17:27 olamundo.sh
-rwxrwxr-x  1 herlon herlon 76 2005-02-13 11:12 olamundo2.sh
-rw-rw-r--  1 herlon herlon 35 2005-02-27 18:28 res_olamundo.sh

[prompt]$ echo "Esta linha está no final do arquivo." >> conteudo.txt
[prompt]$ cat conteudo.txt
total 12
-rw-rw-r--  1 herlon herlon  0 2005-02-27 22:41 conteudo.txt
-rwxrwxr-x  1 herlon herlon 60 2005-02-27 17:27 olamundo.sh
-rwxrwxr-x  1 herlon herlon 76 2005-02-13 11:12 olamundo2.sh
-rw-rw-r--  1 herlon herlon 35 2005-02-27 18:28 res_olamundo.sh
Esta linha está no final do arquivo.

[prompt]$ ls nao_existe 2> erro.txt
[prompt]$ cat erro.txt
ls: nao_existe: Arquivo ou diretório não encontrado

[prompt]$ ls -l mensagem
-rw-rw-r--  1 herlon herlon 30 2005-02-27 22:51 mensagem
[prompt]$ mail herlon < mensagem

[prompt]$ mail herlon << FIM
> Oi Herlon,
> Como vai? Tudo bem?
> FIM
[prompt]$

[prompt]$ cat alunos | sort | lp

[prompt]$ rm naoexiste
rm: cannot lstat ‘naoexiste’: Arquivo ou diretório não encontrado
[prompt]$ rm naoexiste 2> /dev/null
[prompt]$
```

Figura 2.2: Exemplos de uso dos caracteres de redirecionamento.

2.3 COMANDOS ÚTEIS

Uma das grandes dificuldades do iniciante em programação de *shell-scripts* é a falta de conhecimento de alguns comandos do Linux, e suas respectivas listas de opções, que podem simplificar a elaboração de um *script*. Esta seção apresenta os comandos do Linux mais utilizados na elaboração de um *shell-script*.

2.3.1 *grep*, *egrep* e *fgrep*

Grep – Global Regular Expression Print tem como principal objetivo localizar cadeias de caracteres dentro de um texto previamente definido. Esse texto pode ser o conteúdo de um arquivo, a saída de um programa ou a entrada padrão. A cadeia de caracteres é definida por uma Expressão Regular[1].

Na Figura 2.3 é mostrado um exemplo onde `grep` procura pela cadeia de caracteres “Linux” dentro do arquivo `meutexto.txt`. O comando devolve as linhas inteiras que contêm a cadeia procurada. No exemplo da Figura 2.4, `grep` procura pela cadeia “livre” dentro de todos os arquivos com extensão `.txt` no diretório corrente. Ele retorna o nome do arquivo seguido pela linha que contém a cadeia procurada.

```
[prompt]$ grep Linux meutexto.txt
Linux é um sistema operacional multiusuário e
O Linux apresenta interatividade com outros
Linux é um software de livre distribuição, ou
```

Figura 2.3: Exemplo do `grep` procurando *string* dentro de um arquivo especificado.

```
[prompt]$ grep livre *.txt
meutexto.txt:Linux é um software de livre distribuição, ou
teste.txt:Uso o software livre por ser
```

Figura 2.4: Exemplo do `grep` procurando *string* dentro de vários arquivos.

No exemplo da Figura 2.5, `grep` recebe a saída do comando `ps aux` e retorna as linhas que contêm a cadeia “kile”.

```
[prompt]$ ps aux | grep kile
herlon    3443  1.4  5.4 41580 27336 ?       S    09:17   0:20 kile
herlon    3512  0.0  0.1  2916  628 pts/2    R    09:40   0:00 grep kile
```

Figura 2.5: Exemplo do `grep` procurando *string* em saída de comando.

As opções comuns mais utilizadas com `grep` são: `-c` que retorna apenas a quantidade de linhas encontradas; `-l` que retorna apenas os nomes dos arquivos que contêm a cadeia de caracteres procurada; e `-v` que retorna a entrada completa exceto as linhas onde a ocorrência foi encontrada. Os exemplos da Figura 2.6 mostram o uso das opções `-c` e `-l`. No exemplo da Figura 2.7, removeu-se o usuário “fulano” dos arquivos `/etc/passwd` e `/etc/shadow`.

[1] Expressões Regulares serão estudadas no Capítulo 3

```
[prompt]$ grep -c Linux meutexto.txt
3
[prompt]$ grep -l livre *.txt
meutexto.txt
teste.txt
```

Figura 2.6: Exemplo de uso do `grep` com as opções `-c` e `-l`.

```
[prompt]# grep -v fulano /etc/passwd > /etc/passwd.new
[prompt]# grep -v fulano /etc/shadow > /etc/shadow.new
```

Figura 2.7: Exemplo de uso do `grep` com a opção `-v`.

Da mesma família do `grep`, há ainda o `egrep` – *Extended grep* e o `fgrep` – *Fast grep*. O `egrep` é mais poderoso que o `grep`, porém mais lento, sendo utilizado com Expressões Regulares mais complexas. O `fgrep` é mais rápido que o `grep` mas não trabalha com Expressões Regulares.

2.3.2 *wc*

O comando `wc`, quando usado com a opção `-l`, conta o número de linhas da ocorrência. Com a opção `-c`, conta o número de caracteres e com `-w`, o número de palavras. Um exemplo de uso do `wc` pode ser visto na Figura 2.8 onde são contados o número de linhas, caracteres e palavras do texto gravado no arquivo `meutexto.txt`.

```
[prompt]$ cat meutexto.txt | wc -l
8
[prompt]$ cat meutexto.txt | wc -c
363
[prompt]$ cat meutexto.txt | wc -w
54
```

Figura 2.8: Exemplo de uso do comando `wc`.

2.3.3 *cut*

O comando `cut` é usado para extrair pedaços de dados de um arquivo ou da saída redirecionada de um comando. Geralmente, é usado com as opções `-c`, `-f` e `-d`.

A opção `-c` especifica uma porção que se deseja cortar através de posições de caracteres. A Tabela 2.4 mostra detalhes do uso da opção `-c`, e a Figura 2.9 mostra exemplos de uso do `cut` com a opção `-c`.

Nem todos os arquivos e saídas de comando redirecionadas possuem campos de dados em posições fixas em relação a número de caracteres. A opção `-f` serve para

Tabela 2.4: Formato geral do comando `cut` com a opção `-c`.

Comando	***Descrição***
`cut -c`*posição* `[arquivo]`	Retorna todos os caracteres de *posição*.
`cut -c`*pinicio*`-`*pfinal* `[arquivo]`	Retorna todos os caracteres entre *pinicio* e *pfinal*.
`cut -c`*pinicio*`- [arquivo]`	Retorna todos os caracteres após *pinicio*.
`cut -c-`*pfinal* `[arquivo]`	Retorna todos os caracteres do início até *pfinal*.

especificar os campos que serão extraídos. As regras de delimitação são as mesmas da opção `-c`. Mas a opção `-f` sozinha só fará efeito se o separador de campos for o caracter <TAB>. Sendo outro caracter diferente é necessário o uso da opção `-d` para especificar qual é o caracter de separação. Se o caracter de separação for um caracter que o shell possa interpretar como metacaracter, é necessário que este venham entre aspas ou apóstrofos. A Figura 2.10 mostra um exemplo de uso do `cut` com as opções `-f` e `-d`.

2.3.4 *paste*

Ao contrário do `cut` que separa campos, o `paste` permite juntar campos de diferentes arquivos. Entre os campos, `paste` usa o caracter <TAB> para fazer a separação. A opção `-d` pode definir qual será o caracter delimitador. A Figura 2.11 mostra um exemplo de uso do comando `paste`.

2.3.5 *head*

O comando `head` é geralmente usado na forma `head -n`, onde n é um número que representa as “n” primeiras linhas de um arquivo ou de uma saída de comando redirecionada. Um exemplo de seu uso é mostrado na Figura 2.12.

2.3.6 *tail*

O comando *tail* é geralmente usado na forma `tail -n`, funcionando de forma inversa a `head`, retornando as “n” últimas linhas de um arquivo ou de uma saída de comando redirecionada. Um exemplo pode ser visto na Figura 2.13.

2.3.7 *expr*

O comando `expr` trabalha com expressões matemáticas simples e somente números inteiros. Permite, também, manipulação de *strings*, mas com recursos limitados. Exemplos de operações matemáticas com `expr` podem ser vistos na Figura 2.14. Neste texto, o

```
[prompt]$ ls -l
-rw-rw-r--  1 herlon herlon  292 2005-02-27 22:43 conteudo.txt
drwxrwxr-x  2 herlon herlon 4096 2005-03-01 10:46 erro
-rw-rw-r--  1 herlon herlon  363 2005-03-01 09:23 meutexto.txt
-rw-rw-r--  1 herlon herlon   47 2005-03-01 09:58 teste.txt
drwxrwxr-x  2 herlon herlon 4096 2005-03-01 10:46 tmp

[prompt]$ ls -l | cut -c1
-
d
-
-
d

[prompt]$ls -l | cut -c8-10
r--
r-x
r--
r--
r-x

[prompt]$ls -l | cut -c33-
2005-02-27 22:43 conteudo.txt
2005-03-01 10:46 erro
2005-03-01 09:23 meutexto.txt
2005-03-01 09:58 teste.txt
2005-03-01 10:46 tmp

[prompt]$ls -l | cut -c-10
-rw-rw-r--
drwxrwxr-x
-rw-rw-r--
-rw-rw-r--
drwxrwxr-x
```

Figura 2.9: Exemplos de uso do `cut` com a opção `-c`.

tratamento de *strings* é deixado com as ferramentas *Sed* e *Awk* por serem mais completas e eficientes[2].

2.3.8 *bc*

Para expressões numéricas mais complexas, ou com casas decimais, usa-se a calculadora `bc`. Pode-se também ser usada para conversão de base numérica. Exemplos de uso são mostrados na Figura 2.15.

[2] *Sed* e *Awk* são estudadas no Capítulo 3.

```
[prompt]$ cat /etc/passwd | cut -f5 -d:
root
bin
daemon
adm
 ...
Herlon Camargo
Matheus Camargo
Proftpd user (system)
Clam Anti Virus Checker
...
```

Figura 2.10: Exemplo de uso do `cut` com as opções `-f` e `-d`.

```
[prompt]$cat /etc/passwd
...
herlon:x:500:500:Herlon Camargo:/home/herlon:/bin/bash
matheus:x:501:501:Matheus Camargo:/home/matheus:/bin/bash
proftpd:x:101:103:Proftpd user (system):/srv/ftp:/bin/false
httpd:x:502:502::/home/httpd:/bin/bash
postgres:x:503:503::/home/postgres:/bin/bash
teste:x:504:504::/home/teste:/bin/bash
gdm:x:42:42::/home/gdm:/bin/bash
clamav:x:43:43:Clam Anti Virus Checker:/var/lib/clamav:/bin/false
...

[prompt]$ cat /etc/passwd | cut -f1 -d: > /tmp/logins
[prompt]$ cat /etc/passwd | cut -f7 -d: > /tmp/shells

[prompt]$ paste /tmp/logins /tmp/shells
herlon  /bin/bash
matheus /bin/bash
proftpd /bin/false
httpd   /bin/bash
postgres        /bin/bash
teste   /bin/bash
gdm     /bin/bash
clamav  /bin/false
...
```

Figura 2.11: Exemplo de uso do comando `paste`.

2.3.9 *sort*

O comando `sort`, por padrão, ordena os elementos de um arquivo, ou de uma saída de comando redirecionada, em ordem lexicográfica. Ou seja, os elementos são ordenados de acordo com o valor numérico do código ASCII utilizado para representar, na memória,

```
[prompt]$ cat /etc/passwd | head -3
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
```

Figura 2.12: Exemplo de uso do comando `head`.

```
[prompt]$ cat /etc/passwd | tail -3
teste:x:504:504::/home/teste:/bin/bash
gdm:x:42:42::/home/gdm:/bin/bash
clamav:x:43:43:Clam Anti Virus Checker:/var/lib/clamav:/bin/false
```

Figura 2.13: Exemplo de uso do comando `tail`.

```
[prompt]$ expr 3+4
3+4
[prompt]$ expr 3 + 4
7
[prompt]$ expr 5 - 2
3
[prompt]$ expr 10 / 5
2
[prompt]$ expr 2 * 3
expr: erro de sintaxe
[prompt]$ expr 2 \* 3
6
```

Figura 2.14: Exemplos de uso do comando `expr`.

```
[prompt]$ echo "2^2 + 3" | bc
7
[prompt]$ echo "((2+1)^2 + 11)/3" | bc
6
[prompt]$ echo "scale=3; ((2+1)^2 + 11)/3" | bc
6.666
```

Figura 2.15: Exemplo de uso do comando `bc`.

os caracteres que compõem uma *string*. As *strings* são ordenadas como em um dicionário, mas a ordenação de números é falha: por exemplo, `100` é considerado menor que `99`, e portanto seria colocado antes. Para uma ordenação numérica perfeita, usa-se a opção `-n`. A Figura 2.16 possui um exemplo com o seu uso.

```
[prompt]$ cat /etc/passwd | cut -f1 -d: | sort
adm
bin
clamav
daemon
ftp
...
teste
uucp
www
```

Figura 2.16: Exemplo de uso do comando `sort`.

2.3.10 *uniq*

O comando `uniq` remove as linhas duplicadas de um arquivo ou de uma saída de comando redirecionada. É necessário que o arquivo esteja ordenado para que as linhas duplicadas sejam consecutivas dentro do arquivo. Quando usado com a opção `-d` retorna somente as linhas duplicadas. A Figura 2.17 mostra um exemplo do uso.

```
[prompt]$ who
matheus  tty2         Mar  1 12:13
root     tty3         Mar  1 12:13
matheus  tty4         Mar  1 12:13
herlon   :0           Mar  1 08:46
herlon   pts/0        Mar  1 08:46
herlon   pts/1        Mar  1 09:17
herlon   pts/2        Mar  1 09:20
herlon   pts/3        Mar  1 09:24

[prompt]$ who | cut -f1 -d" " | uniq
matheus
root
matheus
herlon

[prompt]$ who | cut -f1 -d" " | sort | uniq
herlon
matheus
root
```

Figura 2.17: Exemplo de uso do comando uniq.

2.4 CARACTERÍSTICAS INICIAIS DO *SHELL-SCRIPT*

As distribuições Linux trazem como padrão o *shell* Bash, geralmente instalado no diretório `/bin/bash`. O *shell* aceita os comandos especificados num *script*, interpreta-os e faz com que o sistema operacional execute os comandos na maneira e na ordem que foram especificados. Um *shell-script* é uma coleção de um ou mais comandos em um arquivo. Para executar esse *script* basta que se digite o nome do arquivo na linha de comandos.

Um *shell-script* aceita, além de todos os comandos que você poderia digitar na linha de comandos, variáveis de ambiente e sistema, variáveis definidas pelo usuário, estruturas de controle de fluxo, saídas formatadas, acesso a banco de dados, e muitos outros recursos. Possuindo várias características e semelhanças com linguagens de programação, permite a elaboração de *scripts* complexos e eficientes para a administração de sistemas. Nas seções seguintes, são apresentados vários recursos para elaboração de *shell-scripts* usando a sintaxe do Bash.

2.4.1 O Primeiro *Script*

O primeiro *script* será o tradicional "*Olá Mundo!*", cujo código é apresentado na Figura 2.18.

```
#!/bin/bash

echo "Olá␣Mundo!"              # Imprime uma frase.
```

Figura 2.18: Código-fonte do *script* `olamundo.sh`

A extensão nos *shell-scripts* é optativa. Recomenda-se utilizar a extensão `.sh` para que o leitor possa facilmente identificar que tipo de arquivo está lidando, sem precisar abri-lo ou visualizá-lo. Outras extensões poderiam ser colocadas no lugar de `.sh`, como por exemplo `.bash` ou outra qualquer. O *script* da Figura 2.18 foi nomeado como `olamundo.sh`.

Pode-se executar esse *script* de duas maneiras. A primeira, seria dar permissão de execução ao arquivo e depois executá-lo diretamente, como pode ser visto na Figura 2.19. A permissão de execução será dada apenas uma vez, antes da primeira execução do *script*.

```
[prompt]$ chmod +x olamundo.sh
[prompt]$ ./olamundo.sh
Olá Mundo!
```

Figura 2.19: Execução do *script* `olamundo.sh` de forma direta.

A segunda maneira seria através da chamada direta do *shell* Bash, como na Figura 2.20, passando o nome do arquivo como parâmetro. Dessa forma, não há necessidade do *script* ter permissão para execução.

```
[prompt]$ /bin/bash olamundo.sh
Olá Mundo!
```

Figura 2.20: Execução do *script* `olamundo.sh` através da chamada do interpretador.

Pelas Figuras 2.18, 2.19 e 2.20, o leitor deve observar que o autor usou dois tipos diferentes de molduras nas figuras: uma para código e outra para o resultado da execução do código. Na moldura de códigos, os espaços em branco serão marcados com um caracter parecendo o caracter "traço-baixo", para alertar o leitor sobre a ocorrência de espaços em branco.

Esse primeiro *script* fornece noções da sintaxe usada em *shell-scripts* desenvolvidos em Bash. A primeira linha do *script* da Figura 2.18 informa ao sistema qual o interpretador que será chamado para executar esse script, e qual a sua localização[3]. Na maioria dos sistemas Linux, o Bash está localizado no diretório `/bin`. Deve-se ajustar essa primeira linha de acordo com o sistema onde o *script* será executado. A primeira linha, dessa forma, se torna necessária e obrigatória apenas se o *script* for executado conforme a primeira maneira descrita anteriormente, ou seja, com permissão de execução e chamado diretamente na linha de comando. Quando o *script* é executado da segunda maneira, ou seja, o nome do interpretador *shell* é invocado na linha de comando, essa primeira linha do *script* passa a ser desnecessária, pois já foi dito ao sistema quem interpretará o *script*. Independentemente da maneira utilizada para execução do *script*, é de bom costume que se deixe sempre a primeira linha indicando o caminho do interpretador que será utilizado. Assim, o *script* poderá ser executado nas duas maneiras sem necessidade de alteração do código.

A sintaxe de Bash é muito semelhante a de outros *shells*, e até mesmo, em alguns aspectos, da sintaxe de Perl (veja Capítulo 4). Comentários em Bash devem ser feitos através do caracter "#". Assim, todo o conteúdo que vier depois desse caracter até o final da linha será ignorado pelo interpretador (com exceção da primeira linha). Para comentar várias linhas seguidas, é necessário que se use o sinal "#" na frente de cada linha.

O comando utilizado para mostrar informações é o `echo`. Essas informações podem estar entre aspas duplas `"` (ver Seção 2.4.2), aspas simples `'` (ver Seção 2.4.3) e/ou apóstrofo invertido `` ` `` (ver Seção 2.4.4). Pode-se usar "caracter de escape" conforme é usado na linguagem C. Para isso, é necessário passar o parâmetro `-e` para o comando `echo`. A Figura 2.21 mostra o script `olamundo.sh` escrito usando caracteres de escape, e o seu

[3] Para detalhes sobre a necessidade do uso de "`#!`" veja [Budlong (1999)].

resultado é apresentado na Figura 2.22. A Tabela 2.5 mostra alguns dos caracteres de escape mais utilizados em Bash.

```
#!/bin/bash

echo -e "\nOlá\t\tMundo\t\t!\n"          # Imprime uma frase.
```

Figura 2.21: Usando caracteres de escape com `echo`.

```
[prompt]$ ./olamundo2.sh

Olá             Mundo           !

[prompt]$
```

Figura 2.22: Execução do *script* `olamundo.sh` com caracteres de escape.

Tabela 2.5: Caracteres de Escape utilizados em Bash.

Caracter	***Descrição***
\a	Aviso sonoro.
\c	Ignora nova linha.
\f	Mudança de página.
\n	Mudança de linha.
\t	Tabulação horizontal.
num	Caracter octal cujo código ASCII é *num*.

Pode-se observar nas Figuras 2.18 e 2.21 que, em Bash, não é necessário que as linhas do *shell-script* terminem com ponto-e-vírgula, ao contrário de linguagens como C e Perl.

2.4.2 Aspas Duplas – (")

Normalmente, as aspas duplas são usadas para exibição de *strings*. Se uma *string* possuir espaços, pode-se cercá-la com aspas duplas (") de forma que o *shell* a interprete como sendo uma entidade só. Quando se coloca um caracter especial entre aspas duplas, o Bash ignora o seu significado, exceto os caracteres $ (dólar), ‘ (apóstrofo invertido) e \ (barra invertida). Por causa disso, o Bash interpretará qualquer variável presente numa *string* que estiver entre aspas duplas (mais detalhes sobre variáveis são mostrados na Seção 2.5). Exemplos do uso de aspas duplas podem ser vistos na Figura 2.23.

```
[prompt]$ echo Inseri espaços           nesta frase
Inseri espaços nesta frase
[prompt]$ echo "Inseri espaços           nesta frase"
Inseri espaços           nesta frase
```

Figura 2.23: Usando aspas duplas.

2.4.3 Aspas Simples ou Apóstrofo – (')

Pode-se cercar uma *string* com aspas simples para impedir que o Bash interprete as variáveis citadas em seu interior. As aspas simples ignoram todos os caracteres, inclusive os caracteres $ e \. Um exemplo pode ser visto na Figura 2.24.

```
[prompt]$ echo "$HOME é o meu diretório padrão."
/home/herlon é o meu diretório padrão.
[prompt]$ echo '$HOME é o meu diretório padrão.'
$HOME é o meu diretório padrão.
```

Figura 2.24: Usando aspas simples.

2.4.4 Apóstrofo Invertido – (`)

O apóstrofo invertido serve para indicar ao Bash que *execute* a *string* delimitada pelo apóstrofo invertido. Normalmente é usado quando se deseja armazenar em uma variável o resultado da execução de um comando ou quando se deseja mostrar na tela uma frase, tendo embutida nela o resultado de um comando. A Figura 2.25 mostra um exemplo de uso do apóstrofo invertido.

```
[prompt]$ echo "O nome deste computador é `uname -n`."
O nome deste computador é donald.minhacasa.
```

Figura 2.25: Usando apóstrofo invertido.

2.4.5 Barra Invertida – (\)

A barra invertida pode ser usada antes de um caracter especial para impedir que o Bash o interprete. É importante salientar que o Bash ignora um, e somente um, caráter após a barra invertida. Se colocada no final da linha, o Bash interpretará como um aviso de continuação de linha, pois a barra invertida anularia o <ENTER> digitado após a mesma. Mas, se entre a barra invertida e o <ENTER> houver algum caracter extra, como um ou mais espaços em branco, talvez o resultado não seja o esperado. Na Figura 2.26 é mostrado um exemplo de uso da barra invertida.

```
[prompt]$ echo "O nome deste computador é \"‘uname -n‘\"."
O nome deste computador é "donald.minhacasa".
```

Figura 2.26: Usando a barra invertida

2.4.6 Parênteses

Tem-se optado por usar uma construção nova no lugar do apóstrofo invertido. É bom frisar que nem todos os *shells* aceitam essa configuração, mas o Bash já permite trabalhar com ela perfeitamente. Um exemplo é mostrado na Figura 2.27. É o mesmo exemplo da Figura 2.25 só que trocando `‘uname -n‘` por `$(uname -n)`.

```
[prompt]$ echo "O nome deste computador é $(uname -n)."
O nome deste computador é donald.minhacasa.
```

Figura 2.27: Usando parênteses.

2.5 VARIÁVEIS

A programação de *scripts* em Bash suporta três tipos de variáveis: variáveis de ambiente, variáveis de sistema (estas duas são comentadas na Seção 2.2.2) e variáveis de usuário.

2.5.1 Variáveis de Usuário

Variáveis de usuário são definidas pelo programador quando o *script* é escrito, e podem ser usadas e modificadas normalmente. Uma das maiores diferenças entre a programação em Bash e outras linguagens de programação é que, em Bash, não há necessidade de se declarar as variáveis antes de seu primeiro uso. Ou seja, não é necessário especificar se a variável é um número ou uma *string*, por exemplo.

O nome de uma variável é iniciado por uma letra ou um sublinhado (_), seguido ou não por quaisquer caracteres alfanuméricos ou caracter sublinhado.

2.5.2 Usando Variáveis

O *script* `olamundo.sh` será modificado inserindo-se uma variável. Essa variável receberá um valor, que nesse caso será uma *string*, e posteriormente será chamada para devolver esse valor numa instrução. A variável `frase` receberá a *string* “Olá Mundo!”. Para que a variável possa retornar seu valor, necessita-se usar o símbolo $ antes de seu nome. A Figura 2.28 mostra o *script* `olamundo.sh` reescrito.

```
#!/bin/bash

frase="Olá␣Mundo!"     # Variável recebe um valor.
echo $frase            # Imprime o valor da variável.
```

Figura 2.28: Exemplo `olamundo.sh` usando uma variável.

Deve-se notar que, em Bash, na atribuição de valor a uma variável não poderão existir espaços em branco ao redor do sinal de atribuição "=". As formas de definição de variáveis mostradas na Figura 2.29 são erradas. Nesses casos, Bash pode interpretar o nome da variável como sendo o nome de um comando do sistema, e o conteúdo de sua atribuição como parâmetro desse suposto comando. A forma correta de se definir uma variável é mostrada na Figura 2.30

```
[prompt]$ a = 2
bash: a: command not found

[prompt]$ a= 2
bash: 2: command not found

[prompt]$ a =2
bash: a: command not found
```

Figura 2.29: Formas erradas se definir uma variável.

```
[prompt]$ a=2
[prompt]$
```

Figura 2.30: Forma correta de se definir uma variável.

A Figura 2.31 mostra o uso de variáveis juntamente com aspas duplas, aspas simples e apóstrofo invertido.

2.5.3 Variáveis Incorporadas

As variáveis incorporadas são variáveis especiais fornecidas pelo Linux, que podem ser usadas para se obter informações importantes dentro do *script*. A Tabela 2.6 mostra essas variáveis com seus significados, e as Figuras 2.32 e 2.33 mostram um exemplo de uso dentro de um *script*.

A Figura 2.33 mostra o *script* `incorporadas.sh` recebendo três parâmetros. O nome do *script* (arquivo) pode ser recuperado pela variável `$0`. O primeiro parâmetro pode ser recuperado pela variável `$1`, o segundo, pela variável `$2`, o terceiro, por `$3`, e assim

```
[prompt]$ a="Uma string"
[prompt]$ echo $a
Uma string

[prompt]$ b=’expr 2 + 3’
[prompt]$ echo $b
expr 2 + 3

[prompt]$ c=‘expr 2 + 3‘
[prompt]$ echo $c
5
```

Figura 2.31: Usando aspas e apóstrofos na definição de variáveis.

Tabela 2.6: Variáveis Incorporadas.

Variável	***Descrição***
`$0`	Contém o nome do *script*.
`$1, $2, ..., $9`	Contém, respectivamente, o primeiro, segundo, ..., nono parâmetros passados na linha de comandos.
`$#`	Contém a quantidade de parâmetros passados.
`$*`	Vetor que contém todos os parâmetros passados.
`$?`	Código de retorno do último comando ou programa executado dentro do *shell-script*.

```
#!/bin/bash

echo "O nome do script é: $0"
echo "O primeiro parâmetro é: $1"
echo "O segundo parâmetro é: $2"
echo "A quantidade de parâmetros é: $#"
echo "O vetor de parâmteros é: $*"
```

Figura 2.32: *Script* utilizando variáveis incorporadas.

sucessivamente até o nono parâmetro `$9`. A partir do décimo parâmetro é necessário o uso de chaves[4], como, por exemplo `${10}`, `${11}`, `${12}`, e assim em diante. Com as variáveis incorporadas, com exceção de `$?`, podemos obter informações do que foi passado na linha de comando.

[4]Pode-se, também, usar as chaves com nomes de variáveis, como por exemplo `${nome}` em vez de `$nome`.

```
[prompt]$ ./incorporadas.sh gato cachorro cavalo
O nome do script é: ./incorporadas.sh
O primeiro parâmetro é: gato
O segundo parâmetro é: cachorro
A quantidade de parâmetros é: 3
O vetor de parâmteros é: gato cachorro cavalo
```

Figura 2.33: Variáveis incorporadas em ação.

Há um comando chamado `shift` que elimina o primeiro argumento dos parâmetros, deixando `$0` intacto, e fazendo com que `$1` receba o valor de `$2`, `$2` receba o valor de `$3`, e assim sucessivamente. A Figura 2.34 mostra um *script* usando o `shift`, e seu resultado é apresentado na Figura 2.35.

```
#!/bin/bash

echo "O primeiro parâmetro é: $1"
shift                                   # Elimina o primeiro parâmetro e
echo "Agora o primeiro parâmetro é: $1"  # coloca o segundo no lugar do primeiro.
```

Figura 2.34: *Script* usando `shift`.

```
[prompt]$ ./troca.sh fulano ciclano beltrano
O primeiro parâmetro é: fulano
Agora o primeiro parâmetro é: ciclano
```

Figura 2.35: Resultado do *script* usando `shift`.

2.6 OPERADORES

Nesta seção são apresentados alguns dos principais operadores usados em Bash. Eles estão relacionados à comparação de dois ou mais argumentos, e foram divididos conforme os dados que são trabalhados.

Para se fazer a comparação em Bash, utiliza-se o comando `test`. A sua sintaxe é aceita de duas formas, com mostrada na Figura 2.36. A segunda forma é mais recente e mais fácil de se entender. Mas, atenção para o fato de que é necessário haver um espaço em branco após o primeiro colchete (`[`) e um espaço em branco antes do último colchete (`]`), conforme mostrado na Figura 2.36.

```
test expressão        # Formato comum.

[ expressão ]         # Formato recente.
```

Figura 2.36: Uso do comando `test`.

2.6.1 Operadores de String

A Tabela 2.7 mostra os operadores mais usados para comparar duas expressões constituídas por *strings*[5]. Na Figura 2.37 são mostrados alguns exemplos de uso desses operadores.

Tabela 2.7: Operadores de *strings*.

Operador	***Descrição***
=	Verifica se duas *strings* são iguais.
!=	Verifica de duas *strings* são diferentes.
-n	Verifica se a *string* **não é** vazia.
-z	Verifica se a *string* **é** vazia.

```
test -n s1       # Retorna verdadeiro se o comprimento de s1 é maior que zero.

test -z s3       # Retorna verdadeiro se o comprimento de s3 é zero.

[ s1 = s2 ]      # Retorna verdadeiro se a string s1 for igual a string s2

[ s1 != s2 ]     # Retorna verdadeiro se a string s1 for diferente da string s2.
```

Figura 2.37: Exemplo dos operadores de *string*.

2.6.2 Operadores de Números

Os operadores mostrados na Tabela 2.8 são usados para comparar dois números. Na Figura 2.38 são mostrados alguns exemplos de uso desses operadores.

Os operadores >, <, >= e <= para fazerem o efeito esperado com números, deverão fazer parte de expressões que estejam entre parênteses duplos. Se estiverem dentro de colchetes simples ou colchetes duplos, usarão ordem alfabética.

[5] É também possível usar os operadores >, <, >= e <= para comparação lexicográfica, mas para isso é recomendável usar Sed, Awk (explicados no Capítulo 3) ou Perl (explicado no Capítulo 4).

Tabela 2.8: Operadores Numéricos.

Operador	*Descrição*
`-eq` ou =	Verifica se dois números são iguais.
`-ne` ou `!=`	Verifica se dois números são diferentes.
`-gt` ou >	Verifica se um número é maior que outro.
`-lt` ou <	Verifica se um número é menor que outro.
`-ge` ou >=	Verifica se um número é maior ou igual a outro.
`-le` ou <=	Verifica se um número é menor ou igual a outro.

```
[ n1 = n2 ]         # Verdadeiro se n1 igual a n2.

[ n1 != n2 ]        # Verdadeiro se n1 diferente de n2.

[ n1 -gt n2 ]       # Verdadeiro se n1 maior que n2.

[ n1 -le n2 ]       # Verdadeiro se n1 menor ou igual a n2.
```

Figura 2.38: Exemplo dos operadores de números.

2.6.3 Operadores de Arquivos

Os operadores mostrados na Tabela 2.9 podem ser usados para se obter informações sobre arquivos. Na Figura 2.39 são mostrados alguns exemplos de uso desses operadores.

Tabela 2.9: Operadores de arquivos.

Operador	*Descrição*
`-f`	Verifica se é um arquivo regular.
`-d`	Verifica se é um diretório.
`-r`	Verifica se o arquivo tem permissão de leitura.
`-w`	Verifica se o arquivo tem permissão de escrita.
`-x`	Verifica se o arquivo tem permissão de execução.
`-s`	Verifica se o nome do arquivo tem um tamanho maior que zero.
`-u`	Verifica se o *bit* SUID está ativo.
`-g`	Verifica se o *bit* SGID está ativo.
`-k`	Verifica se o *sticky bit* está ativo.

```
[ -f "arquivo1" ]         # Verdadeiro se "arquivo1" for um arquivo regular.

[ -d "diretorio1" ]       # Verdadeiro se "diretorio1" for um diretório.

[ -w "arquivo1" ]         # Verdadeiro se "arquivo1" tiver permissão de escrita.
```

Figura 2.39: Exemplo dos operadores de arquivo.

2.6.4 Operadores Lógicos

Os operadores mostrados na Tabela 2.10 são usados para comparar expressões usando as regras da lógica. Na Figura 2.40 são mostrados alguns exemplos de uso desses operadores.

Tabela 2.10: Operadores lógicos.

Operador	***Descrição***
!	Para negar uma expressão lógica.
-a ou &&	Para fazer um *and* lógico em duas expressões.
-o ou ||	Para fazer um *or* lógico em duas expressões.

```
[ ! -r "arquivo" ]
# Verdadeiro se "arquivo" não tiver permissão de letura.

[ -x "arq1" -a -x "arq2" ]
# Verdadeiro se "arq1" e "arq2" forem executáveis.

[ -f "nome1" ] || [ -f "nome2" ]
# Verdadeiro se pelo menos um deles for arquivo regular.
```

Figura 2.40: Exemplo de operadores lógicos.

Bash, como várias outras linguagens de programação, utiliza lógica de curto-circuito. Assim, esses operadores podem ser usados para encurtar algumas linhas de código. Dessa forma é possível pular a avaliação do seu argumento da direita se ficar claro que o argumento da esquerda já forneceu informações suficientes para decidir o valor geral. As Figuras 2.41 e 2.42 mostram exemplos.

No exemplo da Figura 2.41 usou-se o operador "||", que significa "*ou*". No lado esquerdo da expressão é pedido que verifique se o arquivo `file.sh` existe. Se existir, este lado da expressão retornou "Verdadeiro". Com o operador "ou", o resultado final já será "Verdadeiro" independentemente se do lado direito da expressão for "Verdadeiro" ou "Falso". Ou

```
# O lado direito da expressão só será executado se o lado esquerdo for falso.

[ -f "file.sh" ] || echo "Arquivo file.sh não existe"
```

Figura 2.41: Usando || para encurtar linhas de código.

```
# A mensagem só será impressa se o lado esquerdo da expressão for verdadeiro.

[ -k "arquivo" ] && echo "O sticky bit de arquivo está ativo."
```

Figura 2.42: Usando && para encurtar linhas de código.

seja, "*o lado esquerdo já forneceu informações suficientes para decidir o valor geral*". Se o arquivo não existir, o lado esquerdo irá retornar "Falso". Ainda não se pode afirmar o resultado geral da expressão e, então, força-se a execução do lado direito fazendo com que se imprima na tela uma mensagem de erro.

O resultado geral de uma expressão que possui o operador "&&", que significa "*e*", só será "Verdadeiro" se ambos os lados da expressão forem "Verdadeiros". Portanto, se o lado esquerdo for "Falso", nem adianta executar o lado direito, pois o resultado geral já pode ser antecipado: "Falso". Veja exemplo na Figura 2.42.

Se um dos lados da expressão possui mais de uma instrução, poderá ser criado um bloco de instruções[6], compreendidos por chaves ({}).

2.7 ESTRUTURAS DE CONTROLE

Como uma linguagem de programação, Bash também apresenta estruturas condicionais e de repetição. Nas seções seguintes são detalhadas as maneiras como essas estruturas podem trabalhar em Bash.

2.7.1 Estruturas Condicionais

As declarações condicionais são usadas nos *scripts* para decidir qual parte do programa deve ser executada em função de condições especificadas.

if, *else* e *elif*

A instrução `if` avalia uma expressão lógica para tomar uma decisão, e possui o formato apresentado na Figura 2.43.

[6] Um bloco de instruções também poderá ser aberto, por exemplo, 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`, como mostrado na Seção 2.7.

```
if condição
then
    instrução 1
    instrução 2
       ...
else
    instrução 3
    instrução 4
       ...
fi
```

Figura 2.43: Formato básico da estrutura `if`.

Nota-se que não há necessidade de se usar chaves (`{}`) para limitar os blocos "verdadeiro" e "falso", mesmo que cada bloco possua mais de uma instrução, como é o caso da linguagem C. Para concluir a estrutura `if` usa-se o `if` escrito ao contrário: `fi`.

É muito importante saber que em Bash, ao contrário da grande maioria das linguagens de programação, *verdadeiro* vale `0` e *falso* vale `1`. Essa é uma grande causa de erros em *shell-scripts*, necessitando-se assim de muita atenção às estruturas condicionais. Há várias situações em que isso facilita a vida do programador de *scripts* para administração de sistemas. Em Bash, o código de erro retornado por um programa que é bem sucedido é `0`. Se um programa retorna um código diferente de `0` é sinal que algum erro ocorreu. Com isso, pode-se executar um programa no lugar de um teste. Se o resultado do programa for `0` é sinal que o programa foi bem sucedido e a instrução `if` é executada. Se houve algum erro na execução do programa, a instrução `else` será executada.

As Figuras 2.44 e 2.45 apresentam exemplos de uso da instrução `if`.

```
#!/bin/bash

num=$1

if (( $1 > 10 ))
then
    echo "Você␣digitou␣um␣número␣maior␣que␣10."
else
    echo "Você␣digitou␣o␣número␣$1."
fi
```

Figura 2.44: Exemplo simples de uso do `if`.

Existem situações em que há duas ou mais escolhas possíveis quando a condição `if` não for aceita. Nesse caso pode-se usar a função `elif`, que equivale a "else if". As instruções `if` e `elif` são executadas uma por vez, até que uma seja verdadeira ou até que

```
#!/bin/bash

if who | grep $1 > /dev/null     # grep retornará 0 se encontrar algum valor.
then
    echo "$1 está logado."
else
    echo "$1 não está logado."
fi
```

Figura 2.45: Verificando se um usuário está *logado* com `if`.

a condição `else` seja alcançada. Quando atendida uma condição, o bloco correspondente é executado e os demais ignorados. A Figura 2.46 mostra um exemplo de uso dessas instruções.

```
#!/bin/bash

hora=‘date +%H‘

if [ $hora -lt 12 ]
then
    echo "Bom dia, $LOGNAME"
elif [ $hora -lt 18 ]
then
    echo "Boa tarde, $LOGNAME"
else
    echo "Boa noite, $LOGNAME"
fi
```

Figura 2.46: Usando `if` e `elif`.

case

A estrutura `case` permite selecionar uma entre várias opções de ação, baseando-se num valor de uma variável. Deve ser usada no lugar da declaração `if` quando esta tiver um grande número de condições. A Figura 2.47 contém um exemplo de uso do `case`.

Essa estrutura pode ser usada para executar declarações que dependem de um valor isolado ou de uma faixa de valores. Se nenhum valor for encontrado, há a opção de executar o valor padrão “`*`”. A variável definida logo após a instrução `case` tem que casar com alguma das opções em seguida. Se nenhuma opção for escolhida, a opção padrão será executada.

```
#!/bin/bash

opcao=$1

case $opcao in
             Sim | sim ) echo "Você␣concordou!" ;;
             Nao | nao ) echo "Que␣pena!" ;;
                 *     ) echo "Nem␣sim,␣nem␣não";;
esac
```

Figura 2.47: Exemplo de uso da estrutura `case`.

Deve-se observar que as instruções terminam com um duplo ponto-e-vírgula “;;”. Se não for assim, a instrução seguinte será executada junto. Para encerrar a estrutura `case` usa-se `esac`, que é `case` escrito ao contrário.

2.7.2 Estruturas de Repetição

São usadas para repetir uma série de comandos contidos em sua declaração.

for

A estrutura `for` aceita mais de uma forma, que são mostradas através de exemplos.

O primeiro formato é apresentado na Figura 2.48. A variável `i` recebe a lista de valores que vem depois da declaração `in`, sendo um valor de cada vez. Para cada valor da lista, o loop `for` será executado uma vez.

```
#!/bin/bash

for i in 1 2 3 4 5 6 7 8 9
do
   echo "$i"
done
```

Figura 2.48: Primeiro exemplo da estrutura `for`.

O segundo formato é mostrado na Figura 2.49. A variável `lista` contém uma lista de valores. Da mesma maneira que no primeiro formato, `i` receberá, uma a um, os valores de `lista`. Uma variante desse formato pode ser vista na Figura 2.50.

Um terceiro formato é apresentado na Figura 2.51. Aqui não há a instrução `in`. A variável `i` recebe a lista de parâmetros passada para o *script* na linha de comandos. Esse formato é análogo ao mostrado na Figura 2.52.

```
#!/bin/bash

lista='1 2 3 4 5 6 7 8 9'

for i in $lista
do
   echo "$i"
done
```

Figura 2.49: Segundo exemplo da estrutura `for`.

```
#!/bin/bash

for i in `ls`
do
   echo "$i"
done
```

Figura 2.50: Terceiro exemplo da estrutura `for`.

```
#!/bin/bash

for i
do
   echo "$i"
done
```

Figura 2.51: Quarto exemplo da estrutura `for`.

```
#!/bin/bash

for i in $@
do
   echo "$i"
done
```

Figura 2.52: Quinto exemplo da estrutura `for`.

Um quarto formato, que passou a existir a partir da versão 2 do Bash, permite que o `for` seja declarado numa sintaxe parecida com a da linguagem C. A Figura 2.53 mostra um exemplo de uso do `for` dessa maneira.

```
#!/bin/bash

for (( i=1; $i < 10; i++))
do
   echo "$i"
done
```

Figura 2.53: Nova sintaxe para o comando `for`.

while

A estrutura `while` pode ser usada para executar uma série de instruções enquanto uma condição especificada for “verdadeira”. O *loop* termina quando a condição especificada se tornar “falsa” e for verificada. Pode ser que o *loop* não seja nem iniciado se a condição for avaliada “falsa” desde o começo. Também pode ser que o *loop* nunca acabe se a condição especificada nunca se tornar “falsa”. É importante que o leitor tenha em mente que, em Bash, Verdadeiro é `0` e Falso é `1`. O bloco que será executado na estrutura `while` fica entre as instruções `do` e `done`. Na Figura 2.54 há um exemplo de uso do `while`.

```
#!/bin/bash

i=1

while (( $i < 10 ))
do
   echo "$i"
   i=‘expr $i + 1‘
done
```

Figura 2.54: Exemplo de uso do `while`.

until

A estrutura `until` pode ser usada para executar uma série de instruções até que uma condição seja “verdadeira”, ou seja, ao contrário do `while`, ela é executada enquanto uma condição é “falsa”. A partir do momento em que esta condição se torna “verdadeira” e é verificada, o bloco de instruções do `until` deixa de ser executado. O bloco que será executado na estrutura `until` fica entre as instruções `do` e `done`. Um exemplo de uso do `until` é apresentado na Figura 2.55.

```
#!/bin/bash

i=1

until [ $i = 10 ]
do
   echo "$i"
   i=‘expr $i + 1‘
done
```

Figura 2.55: Exemplo de uso do `until`.

break

A instrução `break` deve ser usada quando se deseja interromper a execução de um loop. Quando usado, a execução do script irá para a linha seguinte ao `done`, ou seja, continua após o bloco de instruções onde o `break` se localizava. Se houver vários blocos, um dentro do outro, pode-se usar a seguinte sintaxe: `break [`*n*`]`, onde *n* representa a quantidade de loops mais internos sobre os quais os comandos irão atuar. Quando nada é especificado, *n* vale 1 (valor default). Um exemplo de uso do `break` pode ser visto na Figura 2.56.

```
#!/bin/bash

i=1

while (( $i < 10 ))
do
   if [ $i = 5 ]
   then
       break                    # Interrompe a execução do while.
   fi
   echo "$i"
   i=‘expr $i + 1‘
done

echo "Terminei␣antes␣do␣cinco"
```

Figura 2.56: Exemplo de uso do `break`.

continue

A instrução `continue` é similar a do `break`, mas há uma diferença sutil: quando o interpretador Bash encontra a instrução `continue` ele abandona aquela iteração e começa

outra dentro do *loop*. Ele não sai do *loop* como o `break` faria. Um exemplo de uso do `continue` pode ser visto na Figura 2.57.

```
#!/bin/bash

i=1

while (( $i < 10 ))
do
   if [ $i = 5 ]
   then
       i=‘expr $i + 1‘
       continue                    # Interrompe uma iteração do while.
   fi
   echo "$i"
   i=‘expr $i + 1‘
done

echo "Pulei␣o␣cinco"
```

Figura 2.57: Exemplo de uso do `continue`.

2.7.3 Comando de saída – *exit*

A instrução `exit` pode ser usada para sair de um *shell-script*. Opcionalmente, pode-se usar um número depois do `exit`. Se o *shell-script* corrente tiver sido chamado por outro *shell-script*, o script que o chamou poderá verificar o código e tomar uma decisão de acordo. Por convenção, retorna-se `0` (zero) quando o *script* chegar até o seu final sem apresentar nenhuma condição de erro. Valores diferentes de `0` (zero) podem ser passados para identificar diferentes tipos de problemas. A Figura 2.58 mostra um exemplo de uso da instrução `exit`.

2.8 FUNÇÕES

Da mesma forma que em outras linguagens, Bash também aceita *funções*. Uma função é um pedaço de programa que executa um certo conjunto de instruções que pode ser usado mais de uma vez. Uma função tem o formato conforme apresentado na Figura2.59.

Os parênteses informam ao Bash que uma função está sendo especificada, a as instruções que serão executadas ficam entre chaves (`{}`). Pode-se observar que pelo menos um espaço em branco deve ser colocado entre as instruções e as chaves de início e fim. Quando a função é chamada no corpo do *script*, ela executará todas as instruções que foram colocadas entre as chaves.

```
#!/bin/bash

arquivo="file.txt"

if [ ! -f "$arquivo" ]
then
    echo "$arquivo␣não␣existe"
    exit 1                                  # Retorna código de erro 1
fi

if [ ! -r "$arquivo" ]
then
    echo "$arquivo␣sem␣permissão␣de␣leitura"   # Retorna código de erro 2
    exit 2
fi

cat $arquivo

exit 0                                      # Sucesso! Código de erro 0
```

Figura 2.58: Exemplo de uso do comando `exit`.

```
nome_funcao ()
{
 instrução 1
 instrução 2
 instrução 3
    ...
}
```

Figura 2.59: Estrutura básica de uma função.

Passa-se parâmetros para uma função colocando-os na frente do nome da função, da mesma forma que se faz com um comando na linha de comandos. Para recuperá-los dentro da função é necessário usar as variáveis incorporadas (Seção 2.5.3) $1, $2, $3, . . . , como qualquer outro comando. Um exemplo do uso de funções pode ser visto na Figura 2.60.

No Bash, as funções devem ser declaradas antes de seu uso, porque ele interpreta as linhas do *script* seqüencialmente, do início ao fim. Bash não compila o *script* antes de sua execução, portanto não teria como saber de ante-mão o significado daquela função.

2.9 *SCRIPTS* INTERATIVOS

Uma das formas de se passar informações para um *shell-script* é através de parâmetros no momento em que o *script* é chamado. Dessa forma, não há interatividade. Ao invés

```
#!/bin/bash

saudacao ()                         # Declaração da função saudacao().
{
 echo "Bom␣dia␣$1␣$2!"
}

nome="Herlon"
sobrenome="Camargo"

saudacao $nome $sobrenome           # Passando parâmetros para função saudacao().
```

Figura 2.60: Exemplo de uso de uma função

de se passar uma série de parâmetros, cujo significado precisa ser decorado pelo usuário, pode-se fazer uso de perguntas ao longo do *script* e também de menus. Esta seção apresenta alguns comandos que podem acrescentar interatividade aos *scripts*.

2.9.1 *read*

O comando `read` recebe a próxima linha da entrada (que pode ser a entrada padrão ou qualquer outra definida pelo usuário) e a atribui a uma variável. A Figura 2.61 mostra um exemplo de um *script* sem interação com o usuário, onde a informação é passada através de parâmetros de linha de comandos. Já a Figura 2.62 mostra o *script* reescrito com o uso do `read`. Ambos os scripts fazem a mesma coisa, só que o segundo de uma forma interativa.

```
#!/bin/bash

nome=$1

echo "Olá␣$nome,␣como␣vai?"
```

Figura 2.61: Exemplo de *script* não interativo.

```
#!/bin/bash

echo -n "Digite␣seu␣nome:␣"     # A opção -n serve para o cursor não
read nome                       # mudar de linha.

echo "Olá␣$nome,␣como␣vai?"
```

Figura 2.62: Exemplo de *script* interativo.

No exemplo da Figura 2.62, o comando `read` dá uma pausa no *script* e espera uma entrada do teclado. Quando a tecla <ENTER> é pressionada, a informação digitada pelo teclado é atribuída à variável `nome`, e o script continua. Se durante a pausa for usada a combinação de teclas <Ctrl+d>, o script será finalizado.

O comando `read` também pode ser usado para ler dados de um arquivo. Ele irá ler linha por linha desse arquivo. Deve ser usado em conjunto com o comando `while` e ter a entrada primária redirecionada para o arquivo. A Figura 2.63 mostra a estrutura a ser utilizada. Um exemplo é mostrado na Figura 2.64.

```
while read var_linha        # Cada linha do arquivo "arq_nome" será atribuída
do        ...               # individualmente à variável "var_linha", que po-
   ... $var_linha ...       # de ser trabalhada normalmente dentro do bloco
          ...               # do while.
done < arq_nome
```

Figura 2.63: Lendo linha por linha de um arquivo.

```
#!/bin/bash

while read linha
do
  login=‘echo "$linha" | cut -f1 -d:‘
  nome=‘echo "$linha" | cut -f5 -d:‘
  echo -e "$login\t\t$nome"
done < /etc/passwd
```

Figura 2.64: Usando o `read` para ler linha por linha de um arquivo.

Algumas opções podem ser passadas para o comando `read`. Dentre as mais utilizadas, podemos destacar as mostradas na Tabela 2.11.

2.9.2 *select*

Há ocasiões em que é interessante oferecer um menu para o usuário selecionar uma opção dentre várias. O comando `select` em conjunto com o comando `case` permite a elaboração desse menu de forma simplificada.

A Figura 2.65 mostra o uso do comando `select`. O *prompt* do menu é atribuído à variável de sistema PS3, que será usada como *prompt* para escolha de uma das opções. Define-se uma variável que receberá a opção escolhida e em seguida a lista de opções. Dentro do `select` usou-se o comando `case` para tratar cada opção.

Tabela 2.11: Opções do comando `read`.

Opção	*Descrição*
`-p`	Fornece um *prompt* para fazer a leitura. Exemplo: `read -p "Digite o seu nome:" nome`, exibirá a frase "Digite o seu nome:" e ficará aguardando uma resposta do usuário. A variável `nome` receberá a resposta do usuário.
`-t`	Aguarda um determinado tempo para que o usuário digite uma resposta. Exemplo: `read-t 10 -p "Digite seu nome:" nome || echo "Digite mais rápido"`, aguardará 10 segundos para que o usuário digite seu nome e tecle <ENTER>. Se isso não ocorrer, o comando `read` retorna um erro e o lado direito da expressão é executado.
`-n`	Espera que o usuário digite n caracteres para encerrar o `read`. A execução pode ser interrompida antes dos n caracteres se pressionada a tecla <ENTER>. Exemplo: `read -n 3 -p "Digite o DDD:" codigoDDD`, encerra automaticamente quando o usuário digitar três caracteres.
`-s`	Tudo que for digitado não aparecerá na tela. Ideal para receber senhas. Exemplo: `read -s -p "Digite a senha:" senha_user`, o que for digitado não será ecoado na tela.

Variável $REPLY

No exemplo da Figura 2.65, as opções listadas no menu foram iguais aos valores possíveis passados ao comando `select`. Na maioria das situações, será necessário colocar no menu uma frase maior, descrevendo a respectiva opção. Nesses casos, pode-se usar a variável de sistema `$REPLY`, que irá armazenar a resposta digitada pelo usuário. A Figura 2.66 apresenta o exemplo anterior fazendo uso da variável `$REPLY`.

2.9.3 Prós e Contras da Interatividade

Do ponto de vista do usuário, um *script* interativo é mais "amigável" que um *script* não interativo. Mas, para o administrador de sistemas e/ou de redes, na maioria das vezes, haverá a necessidade se se programar uma determinada tarefa para ser executada posteriormente, sem a presença do administrador por perto. Nessa situação, o *script* deve estar preparado para ter todas as respostas de que precisa, de forma automática, sem interação com o administrador. Para o administrador, o *script* mais "amigável" é aquele que pode ser executado de forma automática.

```
#!/bin/bash

PS3="Estamos em qual estação do ano? "
echo "As Quatro Estações"
select estacao in verao outono inverno primavera sair
do
   case $estacao in
        verao)     echo "Estamos no Verão!"
                   ;;
        outono)    echo "Estamos no Outono!"
                   ;;
        inverno)   echo "Estamos no Inverno!"
                   ;;
        primavera) echo "Estamos no Primavera!"
                   ;;
        sair)      echo "Saindo do programa!"
                   break
                   ;;
        *)         echo "Opção Inválida!"
                   ;;
   esac
done
```

Figura 2.65: Exemplo de uso do `select`.

2.10 EXEMPLO – CONSTRUINDO UMA LIXEIRA: PARTE 1/2

Esta seção apresenta um exemplo mais completo, abrangendo os conceitos apresentados neste capítulo. Será desenvolvido um *script* que implementa uma “Lixeira de Arquivos”, ou seja, um diretório temporário para o usuário deixar seus arquivos antes da remoção definitiva. Este exemplo é baseado em [Neves (2003)].

A lixeira funciona da seguinte forma: para que um arquivo seja enviado para a mesma, deve-se usar o *script* `lixo.sh`. Esse *script* não apaga os arquivos, mas os envia para a lixeira. Para se restaurar um arquivo ou se esvaziar a lixeira, usa-se o *script* `lixeira.sh` (Seção 3.4.1). Serão construídos dois *scripts* nesse exemplo final.

A idéia consiste em:

- criar um *script* capaz de enviar os arquivos especificados pelo usuário para um diretório temporário, onde ficarão armazenados até que sejam definitivamente apagados, ou então, restaurados para seus lugares de origem;
- o diretório temporário (lixeira) será criado, caso não exista, durante a execução do *script* `lixo.sh`;
- a lixeira será o diretório oculto `.lixeira` que ficará no diretório padrão de cada usuário;

```
#!/bin/bash

PS3="Digite␣o␣número␣de␣sua␣opção:␣"
echo "De␣quê␣você␣mais␣gosta?"
select estacao in "Calor␣do␣verão"\
                  "Vento␣do␣outono"\
                  "Frio␣do␣inverno"\
                  "Flores␣da␣primavera"\
                  "Quero␣sair␣deste␣programa"
do
   echo "Você␣escolheu:␣"
   case $REPLY in
        "1") echo "que␣gosta␣do␣verão,␣apesar␣do␣calor."
             ;;
        "2") echo "que␣curte␣o␣outono,␣mesmo␣com␣muito␣vento."
             ;;
        "3") echo "que␣adora␣o␣inverno,␣mesmo␣passando␣frio."
             ;;
        "4") echo "a␣primavera␣por␣causa␣das␣flores."
             ;;
        "5") echo "que␣deseja␣sair."
             break
             ;;
         *)  echo "opção␣inválida!"
             ;;
   esac
done
```

Figura 2.66: Exemplo de uso do `$REPLY`.

- quando um arquivo for enviado para a lixeira, será gravada uma linha no final do arquivo contendo informações para restauração;
- a restauração e/ou esvaziamento da lixeira serão feitos através do *script* `lixeira.sh`, onde, através de um menu, escolhe-se a ação desejada.
- para restaurar um arquivo, o script verifica a última linha do arquivo para recuperar as informações necessárias, e o envia para seu local de origem;
- para esvaziar a lixeira, simplesmente apaga-se todos os arquivos dentro do diretório temporário (lixeira) com o comando `rm`.

O primeiro *script* chamado `lixo.sh` será construído neste capítulo. O segundo *script*, de nome `lixeira.sh` será construído no Capítulo 3, após serem apresentados conceitos de Sed e Awk

2.10.1 Script `lixo.sh`

É importante que o leitor saiba que, por se tratar de um exemplo didático e, apesar de todos os esforços e testes realizados, o autor não pode garantir que o *script* `lixo.sh` funcionará corretamente para todos os tipos de arquivos, podendo, no caso de falhas, haver a perda total de dados dos arquivos manipulados. Recomenda-se que o leitor faça testes antes em arquivos sem importância, até adquirir uma certa confiança no *script* utilizado no exemplo.

Para utilizar o *script* `lixo.sh`, deve-se passar como parâmetros os nomes dos arquivos a serem enviados para a lixeira. Se não houver parâmetros, será mostrada um mensagem de erro. Na Figura 2.67 é mostrada a verificação da chamada do *script*.

```
#!/bin/bash

if [ $# -eq 0 ]
then
    echo "Erro! Use: $0 arquivo1 [arquivo2] [arquivo3] ..."
    echo "          É permitido o uso de metacaracteres."
    exit 1
fi
```

Figura 2.67: Testando os parâmetros do *script* `lixo.sh`.

Em seguida, define-se o diretório que será usado como lixeira, em função do diretório padrão do usuário. Testa-se se o diretório já existe, e em caso negativo será criado. Testa-se também se há permissão de escrita nesse diretório. Em caso negativo será exibida uma mensagem de erro. A Figura 2.68 mostra essa parte do código.

Dentro do loop `for`, verifica-se todos os arquivos que foram passados como parâmetros, quanto a sua existência e se há permissão de movê-los. Um outro teste é feito para que se saiba se estão sendo apagados os arquivos que estão dentro da lixeira. Para apagar arquivos dentro da lixeira existe um *script* apropriado (`lixeira.sh`). Por último, são anexadas, ao final do arquivo, informações para restauração. A Figura 2.69 mostra o loop `for`.

O código completo do script `lixo.sh` pode ser visto nas Figuras 2.70 e 2.71.

```
DIR_LIXEIRA="$HOME/.lixeira"

if [ ! -d $DIR_LIXEIRA ]
then
    mkdir $DIR_LIXEIRA
fi

if [ ! -w $DIR_LIXEIRA ]
then
    echo "Lixeira␣($DIR_LIXEIRA)␣sem␣permissão␣de␣escrita."
    echo "␣␣␣␣␣Mude␣a␣permissão␣e␣tente␣novamente␣..."
    exit 2
fi

erro=0
```

Figura 2.68: Criando o diretório `.lixeira`.

```
for arquivo
do
    if [ ! -f $arquivo ]
    then
        echo "$arquivo␣não␣existe."
        erro=3
        continue
    fi

    DIR_ORIGEM=‘dirname $arquivo‘

    if [ ! -w $DIR_ORIGEM ]
    then
        echo "Sem␣permissão␣de␣remover␣no␣diretório␣$DIR_ORIGEM."
        erro=4
        continue
    fi

    if [ "$DIR_ORIGEM" = "$DIR_LIXEIRA" ]
    then
        echo "Use␣\"lixeira.sh\"␣para␣esvaziar␣a␣lixeira."
        erro=5
        continue
    fi

    cd $DIR_ORIGEM
    echo "" >> $arquivo
    pwd >> $arquivo
    mv $arquivo $DIR_LIXEIRA
    echo "$arquivo␣enviado␣para␣lixeira."
done
exit $erro
```

Figura 2.69: Verificando arquivos que serão apagados.

```
#!/bin/bash

if [ $# -eq 0 ]
then
    echo "Erro! Use: $0 arquivo1 [arquivo2] [arquivo3] ..."
    echo "          É permitido o uso de metacaracteres."
    exit 1
fi

DIR_LIXEIRA="$HOME/.lixeira"

if [ ! -d $DIR_LIXEIRA ]
then
    mkdir $DIR_LIXEIRA
fi

if [ ! -w $DIR_LIXEIRA ]
then
    echo "Lixeira ($DIR_LIXEIRA) sem permissão de escrita."
    echo "     Mude a permissão e tente novamente ..."
    exit 2
fi

erro=0

for arquivo
do
    if [ ! -f $arquivo ]
    then
        echo "$arquivo não existe."
        erro=3
        continue
    fi

    DIR_ORIGEM=‘dirname $arquivo‘

    if [ ! -w $DIR_ORIGEM ]
    then
        echo "Sem permissão de remover no diretório $DIR_ORIGEM."
        erro=4
        continue
    fi
```

Figura 2.70: Código completo do *script* `lixo.sh`.

```
    if [ "$DIR_ORIGEM" = "$DIR_LIXEIRA" ]
    then
        echo "Use \"lixeira.sh\" para esvaziar a lixeira."
        erro=5
        continue
    fi

    cd $DIR_ORIGEM
    echo "" >> $arquivo
    pwd >> $arquivo
    mv $arquivo $DIR_LIXEIRA
    echo "$arquivo enviado para lixeira."
done
exit $erro
```

Figura 2.71: Código completo do *script* `lixo.sh` (continuação).

3

SED, AWK E EXPRESSÕES REGULARES

3.1 INTRODUÇÃO

O objetivo deste capítulo é apresentar ao leitor três ferramentas poderosas para manipulação de textos: Sed, Awk e Expressões Regulares. Elas podem ser utilizadas, tanto diretamente na linha de comando, quanto em arquivos do tipo *script*, e vêm suprir as deficiências do Bash em manipulação de textos.

Documentação sobre o Sed pode ser encontrada em [Jargas (2003)], [Jargas (2005)] e também em [Pizzini (1998)]. Informações detalhadas sobre Awk podem ser encontradas em [FSF (2003)], [Dougherty & Robbins (1997)] e [Robbins (2001)]. E [Friedl (2002)] e [Jargas (2002)] são boas referências sobre Expressões Regulares.

3.2 SED

3.2.1 Características Gerais

O comando sed – *Stream-Oriented Editor*, na realidade, é um editor de textos não interativo, ou seja, orientado por fluxo. A entrada flui pelo sed e é dirigida para a saída padrão. Um exemplo de editores de textos que não são orientados por fluxo seriam o *Vi* e o *Emacs*.

Em geral, a entrada do sed vem de um arquivo ou de um pipe (|), podendo também vir do teclado. A saída vai para a tela por padrão, mas também pode ser guardada em um arquivo ou redirecionada para um outro comando.

Ele não é indicado para ser utilizado como um editor de textos de uso genérico, por não ser nada prático. Simples alterações em arquivos de textos também não são indicadas, pois é mais prático utilizar a função “substituir” de seu editor de textos preferido. Ele é altamente recomendado quando se deseja fazer uma substituição dentro de vários arquivos de texto de uma só vez, ou também, substituição por texto não fixo, que varia em função de algum parâmetro.

A sintaxe para usar o sed tem duas formas, que são mostradas na Figura 3.1. A primeira forma permite que se especifique um comando de edição na linha de comandos, cercado por aspas simples (’). A segunda, permite especificar um *script* contendo comandos sed. Se nenhum arquivo for especificado, sed usará a entrada padrão para ler dados.

```
sed [opções] ’comando’ arquivo(s)

sed [opções] -f script arquivo(s)
```

Figura 3.1: Sintaxe usada com sed.

A Tabela 3.1 mostra as opções do sed.

Tabela 3.1: Opções usadas com sed.

Opção	***Descrição***
-e	Usado para especificar dois ou mais comandos de edição.
-f	Usado para especificar um arquivo contendo comandos de edição.
-n	Usado para que sejam enviadas para a saída somente as linhas que atendam ao critério de pesquisa.

Um arquivo com comandos sed pode-se tornar um arquivo executável, e ser executado diretamente na linha de comandos. Um arquivo desse tipo tem a estrutura conforme mostrada na Figura 3.2. A primeira linha serve para, a exemplo do Bash, indicar o interpretador que irá processar os comandos seguintes. É necessário dar permissão de execução ao arquivo, conforme Figura 3.3.

```
#!/bin/sed

instrução1
instrução2
   ...
```

Figura 3.2: Arquivo de comandos sed.

```
[prompt]$ chmod +x trocaletra.sed
[prompt]$ ./trocaletra.sed
```

Figura 3.3: Executando um arquivo sed.

Os comandos `sed` têm a forma geral mostrada na Figura 3.4, e se constituem em *endereços* e *funções* de edição. As funções consistem em uma única letra ou símbolo. Algumas das mais utilizadas serão mostradas nas seções seguintes. Para uma relação completa dessas funções é recomendável o uso do manual do `sed`[1] Os argumentos dependem de cada função e os endereços definem o escopo de abrangência do comando. Se os dois endereços forem omitidos, a interação será sobre todas as linhas do arquivo especificado. Se somente um for definido, o comando só atuará sobre a linha definida. Um endereço pode ser um número de linha, o símbolo $ que indica última linha ou uma Expressão Regular definida entre barras "`//`". As Expressões Regulares são descritas na Seção 3.5 e também no Capítulo 4.

```
[endereço][,endereço][!]função[argumentos]
```

Figura 3.4: Sintaxe dos comandos usados com `sed`.

Pode-se usar chaves (`{}`) para colocar um endereço dentro de outro ou para aplicar múltiplas funções ao mesmo endereço. No caso de múltiplas funções, a chave de abertura deve estar no final de uma linha e a de fechamento em uma linha sozinha. Deve-se se certificar de que não haja nenhum espaço em branco depois das chaves. A Figura 3.5 mostra o uso de chaves no `sed`.

```
[endereço][,endereço]{
função1
função2
 ...
}
```

Figura 3.5: Usando chaves com `sed`.

O texto mostrado na Figura 3.6, chamado de `meutexto.txt` será utilizado como referência nos exemplos seguintes. O autor aconselha que se crie um arquivo com o conteúdo mostrado na Figura 3.6 para o leitor poder verificar os próximos exemplos.

3.2.2 Substituir – s

A função s substitui a cadeia de caracteres que está entre o primeiro par de barras[2] pela cadeia de caracteres contida no segundo par.

O exemplo mostrado na Figura 3.7 faz a substituição de "Linux" por "GNU-Linux", e apresenta o resultado na tela. Nota-se que o arquivo `meutexto.txt` não foi alterado.

[1] Comando `man sed`.

[2] Sed aceita outros caracteres para delimitação. As barras são os mais largamente utilizados.

```
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.6: Conteúdo do arquivo `meutexto.txt`.

```
[prompt]$ sed 's/Linux/Gnu-Linux/' meutexto.txt
Gnu-Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Gnu-Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Gnu-Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
[prompt]$ cat meutexto.txt
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.7: Substituindo “Linux” por “Gnu-Linux” no arquivo `meutexto.txt`.

Para gravar o resultado em um arquivo, basta redirecionar a saída para esse arquivo. Mas deve-se levar em consideração o seguinte detalhe: não se redireciona a saída do `sed` para o mesmo arquivo de entrada. Quando isso acontece, o shell apagará o arquivo de entrada para poder receber a saída do comando. Isso antes de executar o comando. Portanto, quando o comando for executado, o arquivo de entrada já estará vazio e seu conteúdo estará perdido.

No exemplo da Figura 3.7 observa-se que a última ocorrência da palavra “linux” não foi trocada. Foi pedido para que se trocasse “Linux” e não “linux”. Para que todas as ocorrências sejam satisfeitas, passa-se “`[Ll]inux`” em vez de “Linux”. A Figura 3.8 mostra como ficou essa alteração.

```
[prompt]$ sed 's/[Ll]inux/Gnu-Linux/' meutexto.txt
Gnu-Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Gnu-Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Gnu-Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do Gnu-Linux está
disponível na Internet para os interessados.
```

Figura 3.8: Substituindo "Linux" e "linux" por "Gnu-Linux" no arquivo `meutexto.txt`.

Se em vez de a troca ser por "GNU-Linux" fosse por "GNU/Linux", deveria-se anular a barra para que ela não fosse interpretada. Usa-se, para isso, a contra-barra "\". A Figura 3.9 mostra a troca por "GNU/Linux".

```
[prompt]$ sed 's/[Ll]inux/Gnu\/Linux/' meutexto.txt
Gnu/Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Gnu/Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Gnu/Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do Gnu/Linux está
disponível na Internet para os interessados.
```

Figura 3.9: Usando a contra-barra.

Deseja-se, agora, trocar todas as letras "a" por "u", apenas nas linhas 2 e 3. Se não for colocada a letra g (global) no final, o `sed` trocará apenas a primeira ocorrência em cada linha. Aproveitou-se esse exemplo para redirecionar a saída de um outro comando para o `sed`. A Figura 3.10 mostra como ficou esse exemplo.

```
[prompt]$ cat meutexto.txt | sed '2,3s/a/u/g'
Linux é um sistema operacional multiusuário e
multiturefu que rodu em diversus plutuformus.
O Linux upresentu interutividude com outros
sistemas operacionais. O sistema operacional
Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.10: Definindo endereço para o `sed`.

3.2.3 Imprimir – p

A função p imprime na saída padrão as linhas de um endereço especificado ou que atendam a determinado argumento de pesquisa. A Figura 3.11 apresenta um exemplo onde serão mostradas somente as linhas que contêm a palavra “Linux”. Foi necessário incluir a opção `-n` para que o `sed` mostrasse apenas as linhas requisitadas. A falta dessa opção faz com que o resultado do `sed` mostre todas as linhas do arquivo de entrada, e as linhas que atendem ao parâmetro de pesquisa serão duplicadas. A Figura 3.12 mostra esse mesmo exemplo sem a opção `-n`.

```
[prompt]$ sed -n '/Linux/p' meutexto.txt
Linux é um sistema operacional multiusuário e
O Linux apresenta interatividade com outros
Linux é um software de livre distribuição, ou
```

Figura 3.11: Usando a função p em conjunto com a opção `-n`.

```
[prompt]$ sed '/Linux/p' meutexto.txt
Linux é um sistema operacional multiusuário e
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
O Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Linux é um software de livre distribuição, ou
Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.12: Usando a função p sem a opção `-n`.

Para serem exibidas todas as linhas, exceto as que atendem o argumento de pesquisa, usa-se o símbolo de negação “!” antes de p. Um exemplo é mostrado na Figura 3.13.

```
[prompt]$ sed -n '/Linux/!p' meutexto.txt
multitarefa que roda em diversas plataformas.
sistemas operacionais. O sistema operacional
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.13: Usando o símbolo de negação `!`.

3.2.4 Deletar – d

A função d exclui as linhas de um endereço especificado ou que atendam a determinado argumento de pesquisa. No exemplo mostrado na Figura 3.14 são eliminadas as linhas que contêm a palavra “Linux”. Isso fez o mesmo resultado do exemplo da Figura 3.13.

```
[prompt]$ sed ’/Linux/d’ meutexto.txt
multitarefa que roda em diversas plataformas.
sistemas operacionais. O sistema operacional
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.14: Usando a função d.

3.2.5 Acrescentar – a

Acrescenta um texto após o endereço informado. Deve ser utilizada sempre com a contra-barra “\”. No prompt secundário que se abre digita-se o texto a ser acrescentado. As linhas do texto devem terminar com uma contra-barra, exceto a última. Essa contra-barra esconde o sinal de nova linha. No exemplo da Figura 3.15 insere-se um texto após a quinta linha.

```
[prompt]$ sed ’5a\
> Este texto foi inserido \
> após a quinta linha.’ meutexto.txt
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Linux é um software de livre distribuição, ou
Este texto foi inserido
após a quinta linha.
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.15: Usando a função a.

3.2.6 Inserir – i

Faz o mesmo que a função a (Seção 3.2.5), mas em vez de inserir um texto após o endereço especificado, insere antes do endereço. Veja um exemplo na Figura 3.16.

```
[prompt]$ sed ’5i\
> Este texto foi inserido \
> antes da quinta linha.’ meutexto.txt
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Este texto foi inserido
antes da quinta linha.
Linux é um software de livre distribuição, ou
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.16: Usando a função `i`.

3.2.7 Trocar – c

Semelhante à função a (Seção 3.2.5), só que substitui o endereço especificado por um texto. No exemplo da Figura 3.17 insere-se um texto no lugar da quinta linha.

```
[prompt]$ sed ’5c\
> Este texto foi inserido no \
> lugar da quinta linha.’ meutexto.txt
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Este texto foi inserido no
lugar da quinta linha.
seja, ele pode ser copiado e redistribuído sem
qualquer ônus. O código fonte do linux está
disponível na Internet para os interessados.
```

Figura 3.17: Usando a função `c`.

3.2.8 Finalizar – q

Finaliza a execução do `sed` no endereço especificado ou quando encontrar a primeira ocorrência que atenda a um determinado argumento de pesquisa. O exemplo da Figura 3.18 mostra as cinco primeiras linhas do arquivo, e o exemplo da Figura 3.19 mostra o texto até a linha que contém a palavra “interatividade”.

```
[prompt]$ sed '5q' meutexto.txt
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
sistemas operacionais. O sistema operacional
Linux é um software de livre distribuição, ou
```

Figura 3.18: Finalizando a execução na quinta.

```
[prompt]$ sed '/interatividade/q' meutexto.txt
Linux é um sistema operacional multiusuário e
multitarefa que roda em diversas plataformas.
O Linux apresenta interatividade com outros
```

Figura 3.19: Finalizando a execução ao encontrar a palavra "interatividade".

3.3 AWK

3.3.1 Características Gerais

Awk é um poderoso programa para processamento de arquivos de textos. Os nomes de seus autores[3] deram nome ao programa, que na realidade pode ser considerado uma linguagem de programação direcionada a processamento de textos.

É conveniente usar o Awk para processar um arquivo de texto como se ele fosse composto de registros e campos, como o caso de uma banco de dados textual. Os registros podem ter comprimento fixo ou variável separados por um delimitador, que geralmente é um caracter de nova linha. Os campos do registro são separados também por um caracter delimitador, que por padrão é um espaço em branco ou um <TAB>. Todos esses delimitadores, sejam de registro ou de campo, podem ser alterados previamente.

Entre as características que o Awk tem, como linguagem de programação, pode-se destacar:

- utilizar variáveis de sistema e de usuário;
- utilizar estruturas condicionais e de repetição;
- definição e uso de funções;
- realizar operações aritméticas e de *string*;
- interagir com *shell script*;
- receber argumentos de linha de comando;
- produzir relatórios formatados.

[3] Alfred V. **A**ho, Peter J. **W**einberger, e Brian W. **K**ernighan.

O Awk pode ser invocado diretamente na linha de comandos, ou através de *scripts* contendo vários comandos com a sintaxe do Awk. A implementação GNU do Awk é conhecida como Gawk e pode ser chamada através do comando `gawk` ou mesmo `awk`.

3.3.2 Funcionamento

O Awk pode ser executado de duas formas. A primeira consiste em passar as instruções para o Awk na própria linha de comandos. Essa estrutura é ilustrada na Figura 3.20. As expressões `padrão` e `procedimento` devem vir sempre entre apóstrofos (`' '`). Já a expressão `arquivo(s)` será um ou mais arquivos contendo o texto a ser processado. Essa forma é mais utilizada quando há poucas instruções a serem executadas.

```
awk '[padrão] [{procedimento}]' arquivo(s)
```

Figura 3.20: Passando instruções para o `awk` na linha de comandos.

A segunda forma é mostrada na Figura 3.21. A opção `-f` especifica um arquivo contendo *scripts* em linguagem Awk. Já a expressão `arquivo(s)`, como no caso anterior, será um ou mais arquivos contendo o texto a ser processado. Usa-se um arquivo de *script* quando se possui muitas instruções para serem processadas.

```
awk -f script arquivo(s)
```

Figura 3.21: Passando um *script* para o `awk` processar.

A operação básica do Awk, tanto nas duas formas apresentadas anteriormente, consiste em examinar linha por linha do arquivo de entrada, verificando se alguma atende ao `padrão` e, aquelas que atenderem, executar sobre elas o `procedimento`. Quando não for especificado um arquivo de texto como entrada, o Awk irá ler da entrada padrão.

Para os próximos exemplos apresentados no texto, necessita-se que o autor possua o banco de dados textual apresentado na Figura 3.22. Recomenda-se que o arquivo seja gravado com o nome `estoque.txt`. Esse arquivo contém as informações referentes ao controle de estoque de uma pequena mercearia. Cada registro é um ítem do estoque, e o primeiro campo do registro é o nome da mercadoria, o segundo campo é a quantidade em estoque e, o terceiro, o valor individual de cada mercadoria.

3.3.3 Padrões e Procedimentos

Observa-se que no arquivo `estoque.txt`, os registros estão separados em linhas e os campos estão delimitados com o caracter ponto-e-vírgula “`;`”. Os caracteres separadores de campo padrões do Awk são o espaço em branco ou o caracter <TAB>. Torna-se

```
farinha;36;1.90
oleo de soja;54;2.25
sabonete;85;0.8
detergente;27;1.40
sabao em barra;41;0.35
feijao;16;2.75
arroz;32;2.10
lampada;8;1.64
goiabada;12;4.30
fosforo;26;0.45
```

Figura 3.22: Conteúdo do arquivo `estoque.txt`.

necessário informar ao Awk que o delimitador de campos do arquivo `estoque.txt` é ";". Isso pode ser feito utilizando-se da opção `-F` seguida do caracter delimitador.

No exemplo da Figura 3.23, são listados apenas o nome dos produtos. O leitor pode observar que não foi imposto nenhum `padrão` para ser seguido. Quando isso acontece, o procedimento será aplicado a todas as linhas. No exemplo foi pedido que se listasse apenas o primeiro campo de todos os registros do arquivo `estoque.txt`. Os campos dos registros são representados por `$1`, `$2`, `$3`, respectivamente, para o primeiro, segundo e terceiro campos. O símbolo `$0` representa todos os campos do registro.

```
[prompt]$ awk -F";" '{print $1}' estoque.txt
farinha
oleo de soja
sabonete
detergente
sabao em barra
feijao
arroz
lampada
goiabada
fosforo
```

Figura 3.23: Listando os produtos.

Suponha-se que se deseja listar o nome da mercadoria com o seu respectivo preço. O nome da mercadoria está no primeiro campo e o seu preço está no terceiro campo. A Figura 3.24 mostra esta listagem. O resultado visual pode não ser tão agradável, uma vez que os nomes das mercadorias vêm separados dos preços apenas por um espaço em branco. Na Seção 3.3.4 serão mostradas técnicas de formatação da saída.

```
[prompt]$ awk -F";" '{print $1,$3}' estoque.txt
farinha 1.90
oleo de soja 2.25
sabonete 0.8
detergente 1.40
sabao em barra 0.35
feijao 2.75
arroz 2.10
lampada 1.64
goiabada 4.30
fosforo 0.45
```

Figura 3.24: Listando os produtos com seus respectivos preços.

Para exibir apenas o preço do "feijão", é necessária a definição de um `padrão`, que faça com que o Awk trabalhe apenas com o registro correspondente. A definição desse `padrão` é mostrada na Figura 3.25, juntamente com o resultado da operação.

```
[prompt]$ awk -F";" '$1=="feijao" {print $1,$3}' estoque.txt
feijao 2.75
```

Figura 3.25: Usando padrões para pesquisa.

Os padrões podem ser formados por expressões relacionais, expressões regulares e por padrões especiais.

Expressões Relacionais

As expressões relacionais servem para estabelecer comparações, e seguem às mesmas regras da linguagem C. A Tabela 3.2 mostra os operadores de comparação, e a Tabela 3.3 mostra os operadores lógicos.

As expressões relacionais podem ser usadas com números e com *strings*, e também envolvendo operadores de comparação e lógicos na mesma expressão.

A Figura 3.26 mostra todos os produtos que possuem menos de 20 unidades no estoque, e a Figura 3.27 mostra todos os produtos que começam com a letra "f". Se houvesse apenas a condição `$1 > "f"`, seriam mostrados os nomes de todos os produtos que começam com a letra "f" em diante. Se fosse feito `$1 == "f"` só mostraria produtos que se chamassem exatamente "f".

Tabela 3.2: Operadores de comparação.

Operador	*Descrição*
==	Igual a.
>	Maior que.
<	Menor que.
>=	Maior ou igual a.
<=	Menor ou igual a.

Tabela 3.3: Operadores lógicos.

Operador	*Descrição*
&&	Retorna verdadeiro se os dois lados da expressão forem verdadeiros.
||	Retorna verdadeiro se pelo menos um dos lados da expressão for verdadeiro.
!	Nega a expressão.

```
[prompt]$ awk -F";" '$2 <= 20 {print $1,$2}' estoque.txt
feijao 16
lampada 8
goiabada 12
```

Figura 3.26: Produtos que possuem menos de 20 unidades no estoque.

```
[prompt]$ awk -F";" '$1 >= "f" && $1 <= "g" {print $1}' estoque.txt
farinha
feijao
fosforo
```

Figura 3.27: Produtos que começam com a letra "f".

Expressões Regulares

As expressões regulares são discutidas com mais detalhes na Seção 3.5. É mostrado, nesta seção, apenas um exemplo simples de uso com o Awk.

As expressões regulares devem estar entre um par de barras (`/ /`). O exemplo da Figura 3.27 pode ser reescrito usando expressões regulares. A Figura 3.28 mostra o Awk utilizando expressões regulares na definição do `padrão`. O sinal til (~) significa “corres-

ponde a” e pode ser usado com o sinal de exclamação (`!`) para significar “não corresponde a”. O sinal circunflexo (^) é explicado na Seção 3.5.

```
[prompt]$ awk -F";" '$1 ~ /^f/ {print $1}' estoque.txt
farinha
feijao
fosforo
```

Figura 3.28: Usando expressão regular com o `awk`.

Padrões Especiais – `BEGIN` e `END`

O padrão `BEGIN` permite que se especifique o `procedimento` que será executado antes da primeira linha de entrada ser processada. Muito útil para definição de variáveis globais e apresentação de cabeçalhos.

No padrão `END` pode-se definir o `procedimento` que será executado após todas as linhas da entrada serem processadas. Ideal para apresentar relatórios com valores totais e médios dos registros processados.

A Figura 3.29 mostra o *script* `relatorio1.awk` usando os padrões `BEGIN` e `END`. O cabeçalho da tabela foi definido no `procedimento` do padrão `BEGIN`. O total de mercadoria em estoque foi apresentado no `procedimento` do padrão `END`. O resultado pode ser visto na Figura 3.30.

```
BEGIN { print "Quantidade de produtos" }
 { produtos += $2
   print $1, $2
 }
END { print "Quantidade total de produtos: ", produtos }
```

Figura 3.29: *Script* usando os padrões `BEGIN` e `END`.

3.3.4 Saída Formatada

O exemplo mostrado nas Figuras 3.29 e 3.30 não possui uma saída formatada. Sua apresentação ficou confusa. Para resolver esse problema, pode-se usar o comando `printf`, que possui recursos avançados de formatação. Ele se comporta da mesma forma que na linguagem C, tendo a mesma sintaxe e produzindo os mesmos resultados. A Figura 3.31 mostra o exemplo da Figura 3.29 reescrito usando o comando `printf` para produzir uma aparência melhor. O resultado desse *script*, agora chamado de `relatorio2.awk`, é apresentado na Figura 3.32.

```
[prompt]$ awk -F";" -f relatorio1.awk estoque.txt
Quantidade de produtos
farinha 36
oleo de soja 54
sabonete 85
detergente 27
sabao em barra 41
feijao 16
arroz 32
lampada 8
goiabada 12
fosforo 26
Quantidade total de produtos:  337
```

Figura 3.30: Resultado do *script* `relatorio1.awk`.

```
BEGIN { FS = ";"
        printf "\n%10s %14s %9s\n", "Produto", "Quantidade", "Valor"
        print "==================================" }
 { produtos += $2
   capital += $2*$3
   printf "%-15s %5s %7s %4.2f\n", $1, $2, "", $3
 }
END { print "=================================="
      printf "Quantidade total de produtos: %d\n", produtos
      printf "Capital empregado: R$ %4.2f\n\n", capital }
```

Figura 3.31: *Script* usando saída formatada.

No exemplo da Figura 3.32 a listagem está em ordem alfabética, porque foi utilizado o comando `sort` redirecionando sua saída para o awk, que já recebeu os registros ordenados em função do primeiro campo.

3.3.5 Variáveis

Nota-se no exemplo da Figura 3.32 que não foi passada, na linha de comando, a informação referente ao delimitador de campos. Essa informação foi passada à variável de sistema FS dentro do `procedimento` do padrão `BEGIN`.

O Awk possui outras variáveis de sistema, onde as principais e mais utilizadas são apresentadas na Tabela 3.4. A Figura 3.33 mostra o *script* `relatorio3.awk` fazendo uso das variáveis FNR, NR, além da variável FS. O seu resultado pode ser visto na Figura 3.34

O usuário pode definir suas variáveis e estas não precisam ser declaradas antes de seu primeiro uso. As variáveis podem ser do tipo *string* ou numérica. Dependendo do contexto em que será empregada, o Awk poderá tratá-la como número ou como *string*.

```
[prompt]$ cat estoque.txt | sort | awk -f relatorio2.awk

   Produto     Quantidade     Valor
==================================
arroz              32         2.10
detergente         27         1.40
farinha            36         1.90
feijao             16         2.75
fosforo            26         0.45
goiabada           12         4.30
lampada             8         1.64
oleo de soja       54         2.25
sabao em barra     41         0.35
sabonete           85         0.80
==================================
Quantidade total de produtos: 337
Capital empregado: R$ 497.67
```

Figura 3.32: Resultado do *script* `relatorio2.awk`.

Tabela 3.4: Variáveis de sistema usadas no Awk.

Variável	***Descrição***
`FNR`	Número do registro.
`FS`	Separador de campos utilizado na entrada.
`NF`	Quantidade de campos do registro atual.
`NR`	Quantidade de registros do arquivo de entrada.
`OFS`	Separador de campos utilizado na saída.
`ORS`	Separador de registros utilizado na saída.
`RS`	Separador de registros utilizado na entrada.

Os últimos exemplos já vinham empregando variáveis definidas pelo usuário: `produtos`, `capital` e `valor`.

3.3.6 Funções Internas

O Awk, a exemplo de outras linguagens de programação como C, Perl e Pascal, possui funções pré-definidas já prontas para o usuário utilizar. As tabelas a seguir descrevem

```
BEGIN { FS = ";"
        printf "\n%4s␣␣%10s␣%14s␣%9s\n", "Ítem", "Produto", "Quantidade", "Valor"
        print "======================================"
      }
 { produtos += $2
   capital += $2*$3
   valor += $3
   printf "%-4s␣␣%-15s␣%5s␣%7s␣%4.2f\n", FNR, $1, $2, "", $3
 }
END { print "======================================"
      printf "Quantidade␣total␣de␣produtos:␣%d\n", produtos
      printf "Capital␣empregado:␣R$␣%4.2f\n", capital
      printf "Valor␣médio␣de␣preço:␣R$␣%4.2f\n\n", valor/NR
    }
```

Figura 3.33: *Script* usando variáveis de sistema.

```
[prompt]$ awk -f relatorio3.awk estoque.txt

Ítem     Produto     Quantidade     Valor
======================================
1     farinha            36         1.90
2     oleo de soja       54         2.25
3     sabonete           85         0.80
4     detergente         27         1.40
5     sabao em barra     41         0.35
6     feijao             16         2.75
7     arroz              32         2.10
8     lampada             8         1.64
9     goiabada           12         4.30
10    fosforo            26         0.45
======================================
Quantidade total de produtos: 337
Capital empregado: R$ 497.67
Valor médio de preço: R$ 1.79
```

Figura 3.34: Resultado do *script* `relatorio3.awk`.

resumidamente a maioria das funções utilizadas, divididas conforme suas aplicações. Para maiores informações sobre essas funções, o autor recomenda a leitura do manual do Awk[4].

As funções aritméticas mais utilizadas são mostradas na Tabela 3.5.

A Tabela 3.6 mostra as principais funções para tratamento de *strings*.

[4]Comando `man awk`.

Tabela 3.5: Funções Aritméticas do Awk.

Função	***Descrição***
`sin(x)`	Calcula o seno de `x`.
`cos(x)`	Calcula o cosseno de `x`.
`log(x)`	Calcula o logaritmo natural de `x`.
`exp(x)`	Calcula o exponencial de `x`.
`sqrt(x)`	Calcula a raiz quadrada de `x`.

Tabela 3.6: Funções de *string* do Awk.

Função	***Descrição***
`gsub(expressão,string1,string2)`	Substitui toda `expressão` encontrada na `string2` por `string1`.
`index(substring,string)`	Retorna a posição da `substring` dentro da `string`.
`length(string)`	Retorna o comprimento de `string`.
`match(string,expressão)`	Retorna a posição em `string` onde `expressão` encontra correspondência.
`sub(expressão,string1,string2)`	A primeira `expressão` encontrada em `string2` é substituída por `string1`.
`substr(string,m,n)`	Retorna a subcadeia que começa na posição `m` de `string` com `n` caracteres.

3.3.7 Estruturas Condicionais

As estruturas condicionais `if` e `else` funcionam da mesma forma e com a mesma sintaxe da utilizada na linguagem C. A estrutura básica é mostrada na Figura 3.35. Se a condição testada for verdadeira, as instruções definidas no bloco `if` serão executadas. Se for falsa, serão executadas as instruções do `else`. Se houver apenas uma instrução no bloco, não haverá necessidade de se usar as chaves (`{}`). Um exemplo do uso do `if` e do `else` pode ser visto na Figura 3.36, onde o *script* `barato.awk` informa o produto que tem o menor preço. O resultado da execução do *script* pode ser visto na Figura 3.37.

```
if (expressão)
{
   instrução1
   instrução2
      ...
}
else
{
   instrução1
   instrução2
      ...
}
```

Figura 3.35: Formato básico da estrutura condicional `if-else` no Awk.

```
BEGIN { FS=";"
        minpreco=9999 }
{
   if ( $3 < minpreco )
   {
      barato=$1
      minpreco=$3
   }
}
END { printf "Produto␣mais␣barato␣é␣%s␣que␣custa␣R$␣%3.2f␣\n",barato,minpreco }
```

Figura 3.36: Exemplo de uso da estrutura condicional `if-else`.

```
[prompt]$ awk -f barato.awk estoque.txt
O produto mais barato é o(a) sabao em barra que custa R$ 0.35
```

Figura 3.37: Resultado do *script* `barato.awk`.

3.3.8 Estruturas de Repetição

Assim como as estruturas condicionais (Seção 3.3.7), as estruturas de repetição possuem a mesma forma e sintaxe da utilizada na linguagem C. O formato básico da estrutura de repetição `while` é mostrado na Figura 3.38. As instruções contidas no interior do bloco serão executadas enquanto a condição testada for verdadeira.

Outra estrutura de repetição é o `for`, e o formato básico de sua estrutura é mostrado na Figura 3.39. A `expressão1` inicializa uma variável de *loop*; a `expressão2` impõe uma condição para o *loop* ser executado; e a `expressão3` define como a variável de *loop* será modificada.

O Awk também possui instruções que provocam desvios nas estruturas de repetição. A Tabela 3.7 explica o funcionamento dessas instruções.

```
while (expressão)
{
  instrução1
  instrução2
     ...
}
```

Figura 3.38: Formato básico da estrutura de repetição `while`.

```
for (expressão1; expressão2; expressão3)
{
   instrução1
   instrução2
      ...
}
```

Figura 3.39: Formato básico da estrutura de repetição `for`.

Tabela 3.7: Instruções que provocam desvios nas estruturas de repetição.

Instrução	***Descrição***
`break`	Encerra a execução do *loop*, seja um `for` ou um `while`.
`continue`	Interrompe a iteração atual do *loop*, seja um `for` ou um `while`, começando uma nova iteração.
`exit`	Interrompe a leitura do arquivo de entrada, e executa o padrão `END`, caso exista.
`next`	Pula para o próximo registro.

3.3.9 Vetores

O Awk também permite o uso de vetores. Assim como as variáveis, os vetores não precisam ser declarados antes de seu primeiro uso. A sintaxe dos vetores é parecida com a utilizada pela linguagem Pascal, onde o índice do vetor vem entre colchetes (`[]`). A Figura 3.40 mostra um exemplo da sintaxe utilizada com os vetores.

```
usuario[2] = "fulano"
```

Figura 3.40: Sintaxe utilizada para vetores.

Uma característica importante do Awk é que os índices dos vetores não precisam ser numéricos, podendo ser *strings* também. Isso é semelhante à estrutura `hash` em Perl

(Seção 4.3.3). Um exemplo utilizando os recursos de vetores indexados por valores não numéricos é mostrado na Figura 3.41, onde se construiu um contador de ocorrência de palavras. Ao final da execução, o *script* retorna uma lista com todas as palavras encontradas num texto com a respectiva quantidade de ocorrências. Um arquivo texto é composto de palavras separadas por um espaço em branco. Cada linha do texto é considerada um registro e cada palavra da linha, um campo. O primeiro `for` percorre todas as palavras (campos) da linha (registro), e guarda a informação no vetor `quantidade`, que é indexado por cada palavra diferente encontrada no texto. O segundo `for` imprime o resultado, ordenado numericamente (`-n`) com o comando `sort`, em ordem decrescente (`-r`). A Figura 3.42 mostra o resultado da execução do *script* `contador.sh`

```
{
 for (palavra = 1; palavra <= NF; palavra ++)
     quantidade[$palavra] ++
}
END {
     for (palavra in quantidade)
          print quantidade[palavra], palavra | "sort -nr"
    }
```

Figura 3.41: Exemplo de uso de vetores.

```
[prompt]$ awk -f contador.awk meutexto.txt
3 O
3 Linux
2 é
2 um
2 sistema
2 operacional
2 e
1 ônus.
1 software
1 sistemas
  ...
```

Figura 3.42: Resultado do *script* `contador.awk`.

3.4 EXEMPLO – CONSTRUINDO UMA LIXEIRA: PARTE 2/2

No Capítulo 2, Seção 2.10, deu-se início à construção da “lixeira”, onde foi desenvolvido um *script* de nome `lixo.sh` com a finalidade de mover os arquivos que o usuário informasse para um diretório chamado `.lixeira`, localizado dentro do diretório padrão do

usuário. Antes de mover o arquivo para a “lixeira”, o *script* `lixo.sh` anexava ao arquivo informações úteis para a sua restauração.

A segunda e última parte deste exemplo consiste em elaborar um *script* capaz de exibir a relação dos arquivos na “lixeira”, restaurar um determinado arquivo para a sua origem e esvaziar a “lixeira”.

3.4.1 Script `lixeira.sh`

É importante que o leitor saiba que, por se tratar de um exemplo didático e, apesar de todos os esforços e testes realizados, o autor não pode garantir que os *scripts* `lixo.sh` e `lixeira.sh` funcionarão corretamente para todos os tipos de arquivos, podendo, no caso de falhas, haver a perda total de dados dos arquivos manipulados. Recomenda-se que o leitor faça testes antes em arquivos sem importância, até adquirir uma certa confiança nos *scripts* utilizados no exemplo.

O *script* `lixeira.sh` é executado de forma interativa com o usuário, através de menus. No menu inicial o usuário pode selecionar entre: “Exibir conteúdo”, “Restaurar arquivos”, “Esvaziar lixeira” e “Sair”. A parte do código responsável pela elaboração do menu é apresentada na Figura 3.43.

```
clear
echo -e "\n\t\t\t LIXEIRA \n"
echo -e "\t\tO que você deseja fazer?\n"
PS3="
Digite o número de sua opção: "
select opcao in "Exibir conteúdo"\
                "Restaurar arquivos"\
                "Esvaziar Lixeira"\
                "Sair"
do
    echo -e "\nVocê escolheu $opcao\n"
    case $REPLY in
         "1") exibe
              ;;
         "2") restaura
              ;;
         "3") esvazia
              ;;
         "4") break
              ;;
          * ) echo "Opção inválida!"
              ;;
    esac
done
```

Figura 3.43: Código do menu do *script* `lixeira.sh`.

De acordo com a Figura 3.43, o menu faz referência a três funções: `exibe()`, `restaura()` e `esvazia()`.

A função `exibe()` utiliza o comando `ls -l` para listar o conteúdo da lixeira, e redireciona sua saída para um código em Awk, que formata um relatório para ser exibido na tela. O código da função `exibe()` é mostrado na Figura 3.44.

```
exibe ()
{
  DIR_LIXEIRA="$HOME/.lixeira"
  for arquivo in $(ls $DIR_LIXEIRA)
  do
     DIR_ORIGEM=‘tail -1 $DIR_LIXEIRA/$arquivo‘
     echo "$DIR_ORIGEM"
  done
  ls -l $DIR_LIXEIRA | awk ’BEGIN { printf "%50s\n",
                                    "Conteúdo␣da␣Lixeira\n"
                                 printf "%22s␣%22s␣%10s␣%9s␣%17s\n",
                                    "Arquivo", "Data", "Hora", "Tamanho",
                                    "Origem" }
                           {
                             printf "%35s␣%12s␣%7s␣%9s\n", $8, $6, $7, $5 }’
}
```

Figura 3.44: Código da função `exibe()`.

A função `restaura()` solicita do usuário o nome completo do arquivo a ser restaurado e verifica se realmente o arquivo está presente na “lixeira”. Não havendo erros, é extraída uma informação contida na última linha do arquivo a ser restaurado, informando o nome do diretório de origem. Essa linha foi incluída pelo *script* `lixo.sh` (Seção 2.10.1, página 55). Com o comando `sed` é recriado o arquivo original, sem a última linha que havia sido inserida, e gravado no diretório de origem. O código da função `restaura()` é mostrado na Figura 3.45.

A função `esvazia()` remove todos os arquivos do diretório `.lixeira`, provocando um efeito de “esvaziamento da lixeira”. O código da função `esvazia()` é mostrado na Figura 3.46.

O código completo do *script* `lixeira.sh` pode ser visto nas Figuras 3.47 e 3.48.

3.5 EXPRESSÕES REGULARES

3.5.1 Características Gerais

As Expressões Regulares, também conhecidas como “*regexp*”, são um conjunto de caracteres alfanuméricos e/ou metacaracteres, que combinados entre si de acordo com

```
restaura ()
{
    echo -n "Entre␣com␣o␣nome␣do␣arquivo␣a␣ser␣restaurado:␣"
    read arquivo
    DIR_LIXEIRA="$HOME/.lixeira"
    if [ ! -f $DIR_LIXEIRA/$arquivo ]
    then
        echo -e "\n$arquivo␣não␣existe."
        continue
    fi
    DIR_ORIGEM=‘tail -1 $DIR_LIXEIRA/$arquivo‘
    sed ’$d’ $DIR_LIXEIRA/$arquivo > $DIR_ORIGEM/$arquivo
#    grep -v $DIR_ORIGEM $DIR_LIXEIRA/$arquivo > $DIR_ORIGEM/$arquivo
    rm $DIR_LIXEIRA/$arquivo
    echo -e "\n$arquivo␣restaurado␣em␣$DIR_ORIGEM."
}
```

Figura 3.45: Código da função `restaura()`.

```
esvazia ()
{
    DIR_LIXEIRA="$HOME/.lixeira"
    [ -d $DIR_LIXEIRA ] && { echo -e "Arquivos␣ficarão␣sem␣cópia␣de␣segurança\n"
                             rm -i $DIR_LIXEIRA/*
                            }
    [ -d $DIR_LIXEIRA ] || echo -e "Lixeira␣Vazia!"
}
```

Figura 3.46: Código da função `esvazia()`.

algumas regras, são capazes de extrair informações dentro de textos. São muito utilizadas em conjunto com os comandos da família `grep`, Sed, Awk, Perl, C/C++, editores de textos como Vi, Emac e vários outros. Há pequenas diferenças de sintaxes utilizadas, variando em função da aplicação que está usando as Expressões Regulares. Por exemplo, sua sintaxe no Awk pode diferir ligeiramente da sintaxe utilizada em Sed. Mas em geral, as regras se aplicam a todos os programas, variando apenas a sintaxe.

Este texto enfoca o uso das Expressões Regulares no Awk e no Sed. No Capitulo 4, as Expressões Regulares serão tratadas no contexto de Perl.

3.5.2 Metacaracteres

A Tabela 3.8 mostra os metacaracteres utilizados em Expressões Regulares, seus significados e respectivos exemplos de uso. As Expressões Regulares utilizadas nos exemplos estão no seguinte formato: `/Expressão Regular/`. O leitor pode encontrar informações mais completas em [Friedl (2002)] e [Jargas (2002)].

```
#!/bin/bash

exibe ()
{
  DIR_LIXEIRA="$HOME/.lixeira"
  for arquivo in $(ls $DIR_LIXEIRA)
  do
     DIR_ORIGEM=‘tail -1 $DIR_LIXEIRA/$arquivo‘
     echo "$DIR_ORIGEM"
  done
  ls -l $DIR_LIXEIRA | awk ’BEGIN { printf "%50s\n",
                                          "Conteúdo␣da␣Lixeira\n"
                                  printf "%22s␣%22s␣%10s␣%9s␣%17s\n",
                                          "Arquivo", "Data", "Hora", "Tamanho",
                                          "Origem" }
                             {
                               printf "%35s␣%12s␣%7s␣%9s\n", $8, $6, $7, $5 }’
}

restaura ()
{
    echo -n "Entre␣com␣o␣nome␣do␣arquivo␣a␣ser␣restaurado:␣"
    read arquivo
    DIR_LIXEIRA="$HOME/.lixeira"
    if [ ! -f $DIR_LIXEIRA/$arquivo ]
    then
        echo -e "\n$arquivo␣não␣existe."
        continue
    fi
    DIR_ORIGEM=‘tail -1 $DIR_LIXEIRA/$arquivo‘
    sed ’$d’ $DIR_LIXEIRA/$arquivo > $DIR_ORIGEM/$arquivo
#    grep -v $DIR_ORIGEM $DIR_LIXEIRA/$arquivo > $DIR_ORIGEM/$arquivo
    rm $DIR_LIXEIRA/$arquivo
    echo -e "\n$arquivo␣restaurado␣em␣$DIR_ORIGEM."
}

esvazia ()
{
    DIR_LIXEIRA="$HOME/.lixeira"
    [ -d $DIR_LIXEIRA ] && { echo -e "Arquivos␣ficarão␣sem␣cópia␣de␣segurança\n"
                             rm -i $DIR_LIXEIRA/*
                            }
    [ -d $DIR_LIXEIRA ] || echo -e "Lixeira␣Vazia!"
}
```

Figura 3.47: Código completo do *script* `lixeira.sh`.

Quando usados com o `sed`, os caracteres “`()`” (parênteses) e “`{}`” (chaves) devem vir precedidos por uma “\” (barra invertida).

```
clear
echo -e "\n\t\t\t␣LIXEIRA␣\n"
echo -e "\t\tO␣que␣você␣deseja␣fazer?\n"
PS3="
Digite␣o␣número␣de␣sua␣opção:␣"
select opcao in "Exibir␣conteúdo"\
                "Restaurar␣arquivos"\
                "Esvaziar␣Lixeira"\
                "Sair"
do
    echo -e "\nVocê␣escolheu␣$opcao\n"
    case $REPLY in
         "1") exibe
              ;;
         "2") restaura
              ;;
         "3") esvazia
              ;;
         "4") break
              ;;
          * ) echo "Opção␣inválida!"
              ;;
    esac
done
```

Figura 3.48: Código completo do *script* `lixeira.sh` (continuação).

Um exemplo mais completo e de aplicação real, usando Expressões Regulares, pode ser visto na Seção 4.10, página 118.

Tabela 3.8: Caracteres Especiais em Expressões Regulares

Caracter	***Descrição***
`.`	o “ponto” corresponde a qualquer caracter individual. Ex: `/.ola/` combina com `bola`, `cola` e `2ola`;
*	o “asterisco” corresponde a zero ou mais ocorrências, da Expressão Regular imediatamente anterior. Ex: `/65*/` combina com `6`, `65`, `655`, `6555`, e assim por diante;
+	o “mais” corresponde a uma ou mais ocorrências, da Expressão Regular imediatamente anterior. Ex: `/65+/` combina com `65`, `655`, `6555`, e assim por diante;
?	a “interrogação” corresponde a zero ou uma ocorrência, da Expressão Regular imediatamente anterior. Ex: `/65?/` combina *somente* com `6` e `65`;
^	o “circunflexo” corresponde à Expressão Regular imediatamente posterior, somente se ela estiver no início da linha. Ex: `/^65/` combina com `6543`, `65ar3jt`, se, e somente se, essas expressões estiverem no início da linha;
$	o “dólar” corresponde à Expressão Regular imediatamente anterior, somente se ela estiver no final da linha. Ex: `/65$/` combina com `8765`, `ft43hhv65`, se, e somente se, essas expressões estiverem no final da linha;
\	a “barra invertida” desativa o significado especial do caractere imediatamente posterior. Ex: `/65\+43` combina com `65+43=108`;
`[]`	o “par de colchetes” corresponde com qualquer um dos caracteres incluídos. Um hífen (`-`) indica um conjunto de caracteres consecutivos. Ex: `/[65]/` combina com `6` ou com `5`;
`{n,m}`	o “par de chaves” corresponde a um intervalo de ocorrências da Expressão Regular imediatamente anterior, onde `n` significa o menor número de ocorrências e `m`, o maior número de ocorrências. Se for especificado `{n,}` significa qualquer número de ocorrências acima de `n`. Se for `{,m}` significa um máximo de `m` ocorrências. Ex: `/65{2,4}/` combina com `655`, `6555` e `65555`; `/65{3}/` só combina com `6555`; `/65{,2}/` combina com `6`, `65` e `655`; `/65{2,}/` combina com `655`, `6555`, `65555`, e assim por diante;
`()`	o “par de parênteses”, além de agrupar uma Expressão Regular, salva o texto, correspondido com a Expressão Regular que está no interior dos parênteses, em uma área de armazenamento especial. Até nove padrões podem ser salvos em uma única linha. Eles são recuperados pelas seqüencias de escape `\1` a `\9`. Ex: `/(o)(doce) perguntou para \1 \2 qual era \1 \2 mais \2/` combina com `o doce perguntou para o doce qual era o doce mais doce`;
|	a “barra vertical” corresponde à Expressão Regular especificada imediatamente antes ou depois. Ex: `/senhor(es|as)/` combina com `senhores` e `senhoras`;
`[^]`	o “circunflexo” como primeiro caracter dentro do par de colchetes corresponde a qualquer caracter, exceto os que estão dentro do par de colchetes. Ex: `/[^(janelas)]` corresponde a qualquer palavra exceto `janelas`.

4

PERL

4.1 INTRODUÇÃO

Perl – “Practical Extraction and Report Language” foi criada por Larry Wall na década de 80, sendo uma linguagem que nasceu para facilitar a manipulação de textos, suprindo as necessidades que o Sed, Awk e o *shell* não eram capazes de resolver. É uma linguagem que facilita muito a extração de informações de dentro de arquivos de registro, também conhecidos como arquivos de *log*. Pelos seus recursos e sua flexibilidade, e além de ter o código aberto e estar licenciada nos termos da GPL[1], também é muito utilizada para confecção de páginas *Web*, através de arquivos CGI[2].

Como o objetivo deste texto é o desenvolvimento de ferramentas para automação de tarefas em Linux, será dado enfoque para a extração de informações, ou seja, manipulação de textos. Ao final deste capítulo, o leitor será capaz de desenvolver ferramentas de administração de sistemas, usando recursos da linguagem Perl. O capítulo será finalizado com a criação de uma ferramenta para análise do arquivo de registros de acesso de um servidor *Web*.

O leitor deve verificar se Perl está instalado em seu sistema. Na grande maioria das distribuições Linux ela já vem instalada por padrão. Se caso ela não estiver instalada, o leitor deverá providenciar imediatamente a sua instalação para que possa prosseguir neste capítulo. O site oficial da linguagem é `http://www.perl.org`. Será utilizada neste texto a versão 5.6, que é a última versão estável na data em que este texto foi escrito.

Uma grande vantagem de Perl é a sua comunidade que, assim como a do Linux, colabora com o seu desenvolvimento e uso. O maior exemplo é o CPAN – “Comprehensive Perl Active Network”, que pode ser acessado em `http://www.cpan.org`. Nesse site é possível encontrar uma coleção de módulos e bibliotecas que implementam muitas facilidades à linguagem, além de uma extensa documentação.

---

[1] *General Public License* – `http://www.gnu.org/copyleft/gpl.html`.

[2] *Common Gateway Interface* – são pequenos programas, rodados a partir do servidor, que permitem adicionar vários recursos a uma página *Web*.

Para mais informações sobre a linguagem Perl, o autor recomenda a leitura dos manuais, começando por `man perl`, e também [Wall, et al. (2000)] e [Schwartz & Phoenix (2001)].

4.2 CARACTERÍSTICAS BÁSICAS DE PERL

Normalmente, Perl é utilizada para interpretar um *script*. Antes da execução, o interpretador Perl irá compilar o código, transformando-o numa representação mais eficiente (conhecida como *bytecode*) que será interpretada. Esse esquema permite que, antes de sua execução, seja feita a detecção de erros de sintaxe, além de reduzir o tempo gasto com a execução.

4.2.1 O Primeiro Programa

O primeiro *script* será o tradicional "*Olá Mundo!*", como mostrado na Figura 4.1.

```
#!/usr/bin/perl -w

print "Olá␣Mundo!\n";      # mostra na tela uma mensagem
```

Figura 4.1: Código fonte do *script* `olamundo.pl`.

O *script* foi gravado com o nome de `olamundo.pl`. Na verdade, não é necessário colocar extensão, pois o Linux já reconhecerá o arquivo. Recomenda-se utilizar a extensão `.pl` para que o leitor possa identificar que tipo de arquivo está lidando, sem precisar abri-lo ou visualizá-lo. Poderia ser `.perl` ou qualquer outra.

Há duas maneiras para se executar este *script*. A primeira e mais prática seria dar permissão de execução para o arquivo e depois executá-lo diretamente, como mostrado na Figura 4.2. A permissão de execução será dada somente uma vez, antes da primeira execução.

```
[prompt]$ chmod +x olamundo.pl
[prompt]$ ./olamundo.pl
Olá Mundo!
```

Figura 4.2: Resultado da execução do *script* `olamundo.pl`.

A segunda maneira seria chamar diretamente o interpretador Perl e passar o nome do arquivo como parâmetro. Neste segundo caso, não seria necessário dar permissão de execução ao arquivo. Veja a Figura 4.3.

Este primeiro *script*, apesar de simples, fornece algumas noções sobre a sintaxe de Perl. A primeira linha informa ao sistema qual interpretador será utilizado e só é necessária

```
[prompt]$ perl olamundo.pl
Olá Mundo!
```

Figura 4.3: Executando o *script* `olamundo.pl` através da chamada do interpretador.

se o *script* for executado conforme a primeira maneira discutida anteriormente. Se o *script* for executado de acordo com a segunda maneira, ou seja, chamando diretamente o interpretador, não há necessidade dessa primeira linha. Essa primeira linha deve ser ajustada de acordo com a localização exata do interpretador Perl. Aconselha-se a sempre usar essa linha, pois ela funcionará perfeitamente nas duas maneiras de se executar um programa em Perl. A ausência dessa linha fará com que o *script* só possa ser executado através da chamada direta do interpretador Perl.

Pode-se reparar na semelhança da segunda linha com programas feitos em C. Na verdade, Perl incorpora semelhanças de sintaxe de várias linguagens. Não seria surpresa alguma se Perl estiver parecido com Bash, C, C++ ou outra linguagem conhecida. Comentários podem ser inseridos em um programa através do símbolo "#" (assim como em Bash), e qualquer texto após ele, até o final da linha, será ignorado (com exceção da primeira linha). A única maneira de se usar comentários longos é colocando-se "#" no início de cada linha. Toda instrução em Perl deve terminar com um ponto-e-vírgula, como na segunda linha do *script* `olamundo.pl`. Uma outra semelhança com a linguagem C é o uso de "caracteres de escape". Neste exemplo, foi usado o "\n" para indicar o caracter de mudança de linha. Alguns dos caracteres de escape mais utilizados podem ser vistos na Tabela 4.1. O comando utilizado para exibição de informações na tela foi o `print`. Nas próximas seções são apresentadas as diferenças entre usar aspas duplas `"`, apóstrofo simples `'` e apóstrofo invertido `‘`.

Tabela 4.1: Caracteres de escape em Perl.

Caracter	***Descrição***
\n	Indica mudança de linha.
\r	Cursor retorna ao início da linha
\t	Avança para a próxima posição de tabulação.
\f	Avança para a próxima página (em impressoras).
\b	Cursor retorna um caracter.
\a	Aviso sonoro.

Para que o interpretador Perl forneça avisos de qualquer coisa estranha com o código durante a execução, usa-se a opção `-w` (w de *warnings*) na primeira linha do código (ver Figura 4.4) ou executa-se o interpretador com a opção `-w`, conforme Figura 4.5.

```
#!/usr/bin/perl -w

print "Olá Mundo!\n";      # mostra na tela uma mensagem
```

Figura 4.4: Usando a opção `-w` na primeira linha do código.

```
[prompt]$ perl -w olamundo.pl
```

Figura 4.5: Passando-se a opção `-w` pela linha de comandos.

4.3 VARIÁVEIS EM PERL

Basicamente, os tipos de dados em Perl se distinguem entre singular e plural. *Strings* e números são dados no singular, enquanto que listas de *strings* e listas de números são dados no plural. Na linguagem Perl, variáveis no singular são chamadas de ***scalar*** e variáveis no plural são chamadas de ***array*** e de ***hash***. Variáveis do tipo *scalar* podem receber qualquer tipo de valor escalar: inteiros, números de ponto flutuante, caracter, *string*, e até mesmo referências a outras variáveis. O mesmo ocorre com os *arrays* e *hashs*: pode-se ter vetores de inteiros, de números de ponto flutuante, de caracter, de *string*, ou até mesmo de tipos diferentes, por exemplo, um mesmo vetor contendo números e *strings*.

O exemplo da Figura 4.1 será reescrito utilizando uma variável do tipo *scalar*. Observa-se na Figura 4.6 que não houve a necessidade de se definir com antecedência o tipo da variável `$frase`. Através do símbolo "$" o interpretador Perl saberá que se trata de uma variável do tipo *scalar*, ou seja, contendo um único valor, que neste caso, é uma string. Uma variável do tipo *array* começaria com o símbolo "@". Uma maneira fácil de se guardar qual símbolo usar seria lembrar-se do seguinte: *$calar* e *@rray*. Já o *hash* possui um símbolo não convencional: "%".

```
#!/usr/bin/perl

$frase = "Olá Mundo!\n";   # define uma variável scalar
print $frase;              # mostra na tela uma mensagem
```

Figura 4.6: Usando uma variável do tipo *scalar*.

4.3.1 Variável *scalar*

Variáveis do tipo *scalar* são variáveis que recebem apenas um valor, podendo ser número ou string. Veja na Figura 4.7 alguns exemplos de variáveis desse tipo.

```
$numero = 25;                          # número inteiro
$pi = 3.141593;                        # número real
$tensao = 1.38e4;                      # notação científica
$nome = "fulano";                      # string
$comando = `ls -l`;                    # saída de um comando
$chance = "Hoje␣é␣dia␣do␣$nome.";      # string com substituição
$preco = '0␣preço␣é␣R$2,50';           # string sem substituição
```

Figura 4.7: Exemplos de variável do tipo *scalar*.

O operador "=" atribui o conteúdo da direita à variável da esquerda. As variáveis não precisam ser inicializadas com algum valor. Se for usada uma variável que nunca tenha recebido um valor, essa variável que ainda não foi inicializada passa a existir normalmente. Ela será criada com valor nulo[3], seja `""` ou `0`, dependendo do contexto de onde ela está sendo utilizada.

As variáveis em Perl são interpretadas automaticamente como *strings*, números ou valores booleanos ("*true*" ou "*false*"). Perl também poderá converter automaticamente os dados para o formato necessário ao contexto atual. Veja um exemplo na Figura 4.8.

```
#!/usr/bin/perl

$pontos = "14";              # $pontos é um string.
print $pontos + 5, "\n";     # $pontos age como número.
```

Figura 4.8: Conversão automática do tipo da variável.

O valor original de `$pontos` é uma *string*. Ao ser somada ao número `5`, esta "*string*" passa a responder como número para ser somada, e depois é convertida novamente para *string* para ser impressa como "19".

Na Figura 4.7, pode-se observar que algumas variáveis receberam *strings* entre aspas duplas, entre aspas simples (apóstrofos) e entre apóstrofos inclinados (inclinados à esquerda). Quando o valor está entre aspas duplas poderá haver substituição de variável pelo seu respectivo valor. Quando está entre aspas simples, não poderá haver substituição de variável. E apóstrofos invertidos executarão um programa externo e retornarão a saída do programa, para que se possa apanhá-la como uma única *string* contendo todas as linhas da saída. A Figura 4.9 mostra um exemplo de uso de *strings* e a Figura 4.10 mostra o resultado da saída.

```
#!/usr/bin/perl

$conteudo = ‘ls‘;
print $conteudo;
$animal = "gato";
$preferencia = "Eu␣gosto␣de␣$animal.";
print $preferencia, "\n";
$preferencia = ’Eu␣gosto␣de␣$animal.’;
print $preferencia, ’\n’;
```

Figura 4.9: Usando aspas duplas, simples e apóstrofo invertido.

```
[prompt]$ ./exemplo.pl
barato.awk
contador.awk
conteudo.txt
erro.txt
estoque.txt
     ...
Eu gosto de gato.
Eu gosto de $animal.\n[prompt]$
```

Figura 4.10: Resultado do uso de aspas duplas, simples e apóstrofo invertido.

Observa-se no exemplo da Figura 4.9 que o caracter de escape de final de linha é representado por `"\n"` (entre aspas duplas), já que se deseja mudar de linha. Quando representado por `’\n’`, é impresso uma contra-barra seguida da letra “n”. Ou seja, dessa forma o caracter de escape não tem efeito algum.

4.3.2 Variável *array*

Um *array* é uma lista de vários elementos do tipo *scalar*, indexados pela posição que cada um ocupa na lista. A lista pode conter números, *strings*, ou uma mistura de ambos. A Figura 4.11 mostra vários exemplos de *array*, e pode-se observar como se atribui valores a um *array*, como se manipula individualmente um elemento do *array*, e como se pode definir várias variáveis do tipo *scalar* a partir de um *array*.

Os elementos do *array* são indexados a partir do `0`. Com isso, o primeiro elemento do *array* é referido através do índice `0`. O segundo elemento do *array* é referido pelo índice `1`, e assim sucessivamente. O índice do elemento é sempre um a menos que a posição que ele ocupa dentro do *array*. Este índice é sempre representado entre colchetes “`[ ]`”. Um *array* se comporta como uma pilha, com um início e um final. Perl considera o último elemento do *array* como o topo de uma pilha.

[3] Também chamado de “valor não definido” (*undef*).

```
@fruta = ("banana", "laranja", "maça", "uva");   # array de strings.
@megasena = (4, 15, 16, 25, 42, 48);             # array de números.
@mistura = ("fusca", "telefone", 28);            # array de strings e números.

$animal[0] = "gato";                             # definindo individualmente os
$animal[1] = "cachorro";                         # elementos de um array.
$animal[2] = "girafa";
$animal[3] = "elefante";

$carro = $mistura[0];                            # manipulando um elemento de um array.

($vitamina, $bolo, $torta, $suco) = @fruta;      # definindo escalares a partir
                                                 # de um array.
```

Figura 4.11: Exemplos de variáveis do tipo *array*.

Um recurso muito útil utilizando variáveis de tipo *array* é o *slice* (fatia). Com o *slice* pode-se extrair um pedaço de um *array*. A Figura 4.12 mostra um *script* fazendo o uso do *slice*, e o resultado de sua execução pode ser visto na Figura 4.13. Observa-se que foram extraídos do vetor @a os elementos de índice 2 a 4.

```
#!/usr/bin/perl

@a = ("casa", 23, 35, "fumaça", "avião", 12);
@b = @a[2..4];
print @b[0], "\n";
print @b[1], "\n";
print @b[2], "\n";
```

Figura 4.12: *Script* fazendo uso do *slice*.

```
[prompt]$ ./exemplo.pl
35
fumaça
avião
```

Figura 4.13: Resultado da execução do *script* da Figura 4.12.

O índice do último elemento de um *array* pode ser obtido através do operador "$#". Este operador também pode ser utilizado para alterar a quantidade de elementos de um *array*. Um exemplo é apresentado na Figura 4.14 e o resultado de sua execução pode ser visto na Figura 4.15. Alterando-se para mais, as posições extras recebem um valor "undef". Se alterado para menos, os valores excedentes serão perdidos.

```
#!/usr/bin/perl

@frutas = ("banana", "laranja", "maçã", "uva");
$ind_ult_elem = $#frutas;
print "Número␣de␣elementos␣do␣vetor␣frutas␣(antes):",  $ind_ult_elem + 1, "\n";
$#frutas = 1;
$ind_ult_elem = $#frutas;
print "Número␣de␣elementos␣do␣vetor␣frutas␣(depois):",  $ind_ult_elem + 1, "\n";
```

Figura 4.14: *Script* usando o operador $#.

```
[prompt]$ ./exemplo.pl
Número de elementos do vetor frutas (antes):4
Número de elementos do vetor frutas (depois):2
```

Figura 4.15: Resultado da alteração do tamanho de um *array*.

Para criar um *array* com números inteiros consecutivos e em ordem crescente, usa-se o operador "`..`". A sintaxe é mostrada na Figura 4.16.

```
@dias_mes = 1..31;   # cria um array com 31 elementos numéricos ordenados.
```

Figura 4.16: Criando um *array* com números inteiros e consecutivos.

Pode-se copiar um *array* diretamente para outro. Se eles tiverem tamanhos diferentes, o *array* que receber os valores passará a ter o mesmo tamanho do outro *array*.

Um *array* pode ter a ordem de seus elementos invertida, através da função `reverse()`. Nota-se que esta função não inverte a ordem do *array* original, e sim retorna um novo *array* com a ordem dos elementos invertida. A Figura 4.17 mostra um *script* fazendo uso da função `reverse()` e a Figura 4.18 mostra o resultado de sua execução.

```
#!/usr/bin/perl

@frutas = ("banana", "laranja", "maçã", "uva");
@saturf = reverse @frutas;
print @saturf, "\n";
```

Figura 4.17: *Script* usando a função `reverse()`.

Uma outra função parecida com a `reverse()` é a `sort()` que permite ordenar os elementos de um *array* por ordem lexicográfica[4]. Por isso, ela não deve ser usada com

[4]Os elementos são ordenados de acordo com o valor numérico do código ASCII utilizado para representar, na memória, os caracteres que compõem uma *string*. As *strings* são ordenadas como em um dicionário.

```
[prompt]$ ./exemplo.pl
uvamaçãlaranjabanana
```

Figura 4.18: Resultado da execução do *script* da Figura 4.17.

números, somente com *strings*. Da mesma forma que `reverse()`, `sort()` retorna um novo *array* com os elementos em ordem. A Figura 4.19 mostra um *script* fazendo uso da função `sort()` e a Figura 4.20 mostra o resultado de sua execução.

```
#!/usr/bin/perl

@animais = ("gato", "cachorro", "girafa", "elefante");
@ordem = sort @animais;
print "Animais␣em␣ordem:␣",@ordem, "\n";
```

Figura 4.19: *Script* usando a função `sort()`.

```
[prompt]$ ./exemplo.pl
Animais em ordem: cachorroelefantegatogirafa
```

Figura 4.20: Resultado da execução do *script* da Figura 4.19.

Como Perl trata um *array* como se fosse uma pilha, há funções para inserir e remover elementos no topo e na sua base. As funções `push()` e `pop()` inserem e removem, respectivamente, um elemento na última posição de uma pilha (*array*). Já as funções `unshift()` e `shift()` fazem o mesmo, inserem e removem, respectivamente, só que na base da pilha (*array*). As funções `push()` e `unshift()` podem receber uma lista de elementos para inserir num *array*. Todos os elementos serão inseridos no final ou início, respectivamente, da lista. Um exemplo é mostrado na Figura 4.21.

```
#!/usr/bin/perl

@frutas = ("banana", "laranja", "maçã", "uva");
push @frutas, "goiaba";                         # insere goiaba na última posição.
unshift @frutas, ("pêra", "ameixa");            # insere pêra e ameixa antes
                                                # de banana.
```

Figura 4.21: *Script* usando as funções de inserção e remoção de elemento num *array*.

4.3.3 Variável *hash*

Um *hash* é um conjunto não ordenado de valores do tipo *scalar*, onde seus elementos são indexados por uma *string*. No caso do *array*, os elementos são indexados por números,

onde esses números representam a posição que o elemento ocupa dentro do *array* menos 1. No *hash*, os elementos não tem posição definida, e a única ordem existente é que cada elemento possui sua respectiva chave do tipo *string*. Não poderá haver duas chaves iguais.

Um *hash* não possui início ou fim, ficando, dessa forma, impossível utilizar as funções `push()`, `pop()`, `unshift()` e `shift()`. Para que sejam fornecidos valores a um *hash*, precisa-se informar também a sua chave. As Figuras 4.22, 4.23 e 4.24 mostram exemplos de atribuições de valores a um *hash*.

```
%mes = ("janeiro","jan","fevereiro","fev","março","mar","abril","abr",
        "maio","mai","junho","jun","julho","jul","agosto","ago",
        "setembro","set","outubro","out","novembro","nov","dezembro","dez");
```

Figura 4.22: Criando um *hash* a partir de uma lista.

```
%mes = ( "janeiro"   => "jan",
         "fevereiro" => "fev",
         "março"     => "mar",
         "abril"     => "abr",
         "maio"      => "mai",
         "junho"     => "jun",
         "julho"     => "jul",
         "agosto"    => "ago",
         "setembro"  => "set",
         "outubro"   => "out",
         "novembro"  => "nov",
         "dezembro"  => "dez" );
```

Figura 4.23: Usando o sinal => para se criar um *hash*.

A Figura 4.22 cria um *hash* a partir de uma lista de valores. O primeiro elemento da lista torna-se chave do segundo elemento, o terceiro elemento será a chave do quarto, e assim em diante. Esse é um método de criação de *hash* difícil de se ler. Usando o sinal "=>" como na Figura 4.23, torna a leitura da operação de criação de um *hash* mais fácil. Pode-se também fornecer os pares individualmente, como mostrado na Figura 4.24. O *hash* `%mes` contém a abreviação do nome dos meses, com três letras, e como chaves, os nomes dos meses por extenso.

A Figura 4.25 mostra um exemplo utilizando *hash*, onde se pode observar a extração de valores de um *hash*. Dada uma chave, obtém-se um valor, e nunca o contrário. Observa-se o uso do sinal $ por se tratar de um *scalar*, e não %, que indicaria um *hash* inteiro. O resultado desse *script* pode ser visto na Figura 4.26.

```
$mes{ "janeiro" } = "jan";
$mes{ "fevereiro" } = "fev";
$mes{ "março" } = "mar";
$mes{ "abril" } = "abr";
$mes{ "maio" } = "mai";
$mes{ "junho" } = "jun";
$mes{ "julho" } = "jul";
$mes{ "agosto" } = "ago";
$mes{ "setembro" } = "set";
$mes{ "outubro" } = "out";
$mes{ "novembro" } = "nov";
$mes{ "dezembro" } = "dez";
```

Figura 4.24: Criando um *hash* através de entradas individuais.

```
#!/usr/bin/perl

$idade{ "fulano" } = 35;
$idade{ "ciclano" } = 28;
$idade{ "beltrano" } = 42;

print "A␣idade␣de␣fulano␣é␣$idade{fulano}␣anos.\n";
print "A␣idade␣de␣ciclano␣é␣$idade{ciclano}␣anos.\n";
print "A␣idade␣de␣beltrano␣é␣$idade{beltrano}␣anos.\n";
```

Figura 4.25: Exemplo de uso da variável *hash*.

```
[prompt]$ ./exemplo.pl
A idade de fulano é 35 anos.
A idade de ciclano é 28 anos.
A idade de beltrano é 42 anos.
```

Figura 4.26: Resultado da execução do *script* da Figura 4.25.

Pode-se transformar *hash* em *array* e vice-versa. A Figura 4.27 mostra um exemplo de um *hash* se transformando em *array* e o resultado é exibido na Figura 4.28.

A função `keys()` fornece uma lista das chaves de um *hash*. Já a função `values()` retorna uma lista com os valores. A função `exists()` verifica se há uma determinada chave no *hash*, retornando um valor booleano. A função `delete()` apaga a chave e o seu respectivo valor. A Figura 4.29 mostra um exemplo com o uso dessas funções.

Se uma variável *hash* não tiver valores repetidos, é possível inverter as *chaves* com os *valores*, através da função `reverse()`. Um exemplo é apresentado na Figura 4.30.

```
#!/usr/bin/perl

$idade{ "fulano" } = 35;
$idade{ "ciclano" } = 28;
$idade{ "beltrano" } = 42;

@vetor_idade = %idade;

print "Primeiro elemento de vetor_idade é $vetor_idade[0].\n";
print "Segundo elemento de vetor_idade é $vetor_idade[1].\n";
print "Terceiro elemento de vetor_idade é $vetor_idade[2].\n";
print "Quarto elemento de vetor_idade é $vetor_idade[3].\n";
print "Quinto elemento de vetor_idade é $vetor_idade[4].\n";
print "Sexto elemento de vetor_idade é $vetor_idade[5].\n";
```

Figura 4.27: Transformando *hash* em *array*.

```
[prompt]$ ./exemplo.pl
Primeiro elemento de vetor_idade é beltrano.
Segundo elemento de vetor_idade é 42.
Terceiro elemento de vetor_idade é ciclano.
Quarto elemento de vetor_idade é 28.
Quinto elemento de vetor_idade é fulano.
Sexto elemento de vetor_idade é 35.
```

Figura 4.28: Resultado da execução do *script* da Figura 4.27.

4.4 OPERADORES

Esta seção apresenta alguns dos principais operadores utilizados em Perl. Eles estão classificados de acordo com os dados manipulados. A maioria deles é similar aos utilizados em C, Sed ou Awk.

4.4.1 Operadores Aritméticos

Os operadores aritméticos realizam operações matemáticas, de acordo com as regras matemáticas, ou seja, são avaliados na ordem de precedência conhecida na matemática. Por exemplo: primeiro exponenciação, depois multiplicação e por último soma. Pode-se usar parênteses para alterar esta ordem. A Tabela 4.2 mostra os principais operadores com respectivos exemplos.

```
#!/usr/bin/perl

%mes = ( "janeiro"   => "jan",
         "fevereiro" => "fev",
         "março"     => "mar",
         "abril"     => "abr",
         "maio"      => "mai",
         "junho"     => "jun",
         "julho"     => "jul",
         "agosto"    => "ago",
         "setembro"  => "set",
         "outubro"   => "out",
         "novembro"  => "nov",
         "dezembro"  => "dez" );

@mes_extenso = keys %mes;         # Extrai as chaves do hash.
print "@mes_extenso\n";

@mes_abreviado = values %mes;     # Extrai os valores do hash.
print "@mes_abreviado\n";

$simnao = exists $mes{maio};      # Retorna verdadeiro se a chave maio existe.
print "$simnao\n";

delete $mes{dezembro};            # Elimina a chave com seu respectivo valor.
@sem_dezembro = %mes;             # Converte hash em array.
print "@sem_dezembro\n";
```

Figura 4.29: Usando as funções `keys()`, `values()`, `exists()` e `delete()`.

Tabela 4.2: Operadores Aritméticos.

Operador	*Nome*	*Exemplo*
+	Adição	$a + $b
-	Subtração	$a - $b
*	Multiplicação	$a * $b
/	Divisão	$a / $b
%	Resto de divisão	$a % $b
**	Exponenciação	$a ** $b

```
#!/usr/bin/perl

%mes = ( "janeiro"   => "jan",
         "fevereiro" => "fev",
         "março"     => "mar",
         "abril"     => "abr",
         "maio"      => "mai",
         "junho"     => "jun",
         "julho"     => "jul",
         "agosto"    => "ago",
         "setembro"  => "set",
         "outubro"   => "out",
         "novembro"  => "nov",
         "dezembro"  => "dez" );

%inv_mes = reverse %mes;

print "A␣abreviatura␣de␣$inv_mes{mai}␣é␣$mes{maio}.\n";
```

Figura 4.30: Invertendo chaves com valores numa variável *hash*.

4.4.2 Operadores de *String*

Existem também operadores de "adição" e "multiplicação" de *strings*. Na realidade há *concatenação* e *repetição* de *strings*. A Tabela 4.3 mostra os operadores de *string* com respectivos exemplos.

Tabela 4.3: Operadores de string.

Operador	*Nome*	*Exemplo*
`.`	Concatenação	`$a = "Olá " e $b = "Mundo!", $a.$b = "Olá Mundo!"`
`x`	Repetição	`$a = "bi! " e $b = 3, $a x $b = "bi! bi! bi!"`

O operador de repetição (x) pode parecer sem muito valor, mas em certas ocasiões é muito útil. Um exemplo pode ser visto na Figura 4.31 e seu resultado na Figura 4.32.

4.4.3 Operadores de Atribuição

O operador de atribuição "=" tem sido utilizado em quase todos os exemplos. Ele representa uma atribuição simples, ou seja, determina o valor da expressão no seu lado direito e depois atribui esse valor à variável no lado esquerdo. Ele equivale ao "`:=`" no Pascal. Não se pode confundir este operador de atribuição com o operador de igualdade

```
#!/usr/bin/perl

$n = 40;
print "-*" x $n , "\n";
```

Figura 4.31: Utilizando o operador de repetição.

```
[prompt]$ ./exemplo.pl
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

Figura 4.32: Resultado da execução do *script* da Figura 4.31.

"==". O operador "==" retorna um valor booleano enquanto "=" atribui valor a uma variável. Na Figura 4.33 pode-se observar exemplos do uso deste operador.

```
#!/usr/bin/perl

$i = 5;         # Atribuindo 5 à variável i.

$i = $i + 1;    # Somando 1 à variável i e atibuindo o resultado à
                # própria variável i.
```

Figura 4.33: Utilizando operadores de atribuição.

Nota-se na última expressão do exemplo da Figura 4.33 que a variável "`$i`" foi chamada duas vezes, uma de cada lado do operador. Existem atalhos em Perl, semelhantes aos utilizados em C, que simplificam esta expressão. A Tabela 4.4 mostra alguns exemplos desses atalhos e seus resultados.

Tabela 4.4: Atalhos para operadores de atribuição.

Exemplo	***Descrição***
`$a += 2`	O mesmo que `$a = $a + 2`
`++$a` ou `$a++`	Soma 1 a `$a`
`-$a` ou `$a-`	Subtrai 1 de `$a`
`$b = $a++`	O valor de `$a` é atribuído a `$b`, depois que `$a` é incrementado.
`$b = ++$a`	Primeiro `$a` é incrementado e somente depois é atribuído a `$b`.

4.4.4 Operadores Lógicos

Perl, como várias outra linguagens de programação, utiliza lógica de curto-circuito. Assim, os operadores lógicos podem ser utilizados para encurtar algumas linhas de código. Dessa forma, é possível pular a avaliação do seu argumento da direita se ficar decidido que o argumento da esquerda já forneceu informações suficientes para decidir o valor geral. Funciona da mesma forma que em C e em Bash.

O exemplo da Figura 4.34 usa o operador "`or`", que significa "*ou*". No lado esquerdo da expressão é pedido que se abra o arquivo[5] `file.pl`. Se tudo der certo, este lado da expressão retornou "Verdadeiro". Por estar usando o operador que significa "ou", o resultado final já será "Verdadeiro" independentemente se do lado direito da expressão der "Verdadeiro" ou "Falso". Ou seja, "*o lado esquerdo já forneceu informações suficientes para decidir o valor geral*". Se o arquivo não conseguiu ser aberto, o lado esquerdo irá retornar "Falso". Ainda não podemos afirmar o resultado geral da expressão e, então, força-se o lado direito fazendo com que se imprima na tela uma mensagem de erro.

```
open /home/fulano/file.pl or die "Impossível␣abrir␣o␣arquivo!␣\n"
```

Figura 4.34: Utilizando operadores lógicos.

O resultado geral de uma expressão que possui o operador "`and`", que significa "*e*", só será "Verdadeiro" se ambos os lados da expressão forem "Verdadeiros". Portanto, se o lado esquerdo for "Falso", nem adianta executar o lado direito, pois o resultado geral já pode ser antecipado: "Falso". A Tabela 4.5 mostra os operadores lógicos no contexto de "curto-circuito".

Tabela 4.5: Operadores Lógicos no contexto de "curto-circuito".

Exemplo	***Nome***	***Descrição***
`$a && $b` , `$a and $b`	And	$a se $a for falso, $b caso contrário
`$a || $b` , `$a or $b`	Or	$a se $a for verdadeiro, $b caso contrário
`! $a` , `not $a`	Not	verdadeiro se $a não for verdadeiro
`$a xor $b`	Xor	verdadeiro se $a ou $b for verdadeiro, mas não ambos

[5] Detalhes sobre abertura de arquivos são mostrados na Seção 4.6.

4.4.5 Operadores de Comparação

Os operadores de comparação informam como dois valores do tipo *scalar* se relacionam entre si. Existem operadores de comparação para números e para strings. A Tabela 4.6 mostra os operadores de comparação.

Tabela 4.6: Operadores de comparação.

Número	***String***	***Descrição***
`==`	`eq`	Igual
`!=`	`ne`	Diferente
`<`	`lt`	Menor
`>`	`gt`	Maior
`<=`	`le`	Menor ou igual
`>=`	`ge`	Maior ou igual
`<=>`	`cmp`	`0` se igual, `1` se maior, e `-1` se menor

4.4.6 Operadores de Teste de Arquivo

Os operadores de teste de arquivo permitem que se teste os atributos de um arquivo antes de serem utilizados. Por exemplo, pode-se verificar se um arquivo existe, se é executável, se pode ser gravado, entre outras coisas. Alguns desses operadores estão listados na Tabela 4.7.

4.5 ESTRUTURAS DE CONTROLE

4.5.1 Estruturas Condicionais

if, *else* e *elsif*

A instrução `if` avalia uma expressão booleana e executa um bloco de instruções se a condição for verdadeira. Esse bloco contém uma ou mais instruções agrupadas por chaves "`{}`". Essas chaves sempre serão obrigatórias, mesmo se houver apenas uma instrução no bloco. Em C é opcional o uso de chaves para blocos de apenas uma instrução.

Em caso da condição `if` não for verdadeira, pode-se utilizar a instrução `else`. Nesse caso, o bloco seguinte à instrução `else` será executado.

Existem situações em que há duas ou mais escolhas possíveis quando a condição `if` não for aceita. Neste caso pode-se usar a função `elsif`, que equivale a "*else if*". As

Tabela 4.7: Operadores de teste de arquivo.

Operador	*Nome*	*Exemplo*
`-e $arq`	Exists	Verdadeiro se $arq existir.
`-f $arq`	File	Verdadeiro se $arq for um arquivo regular.
`-d $arq`	Directory	Verdadeiro se $arq for um diretório.
`-r $arq`	Readable	Verdadeiro se $arq puder ser lido.
`-w $arq`	Writable	Verdadeiro se $arq puder ser gravado.
`-x $arq`	Executable	Verdadeiro se $arq for executável.
`-l $arq`	Link	Verdadeiro se $arq for um *link* para outro arquivo.
`-T $arq`	Text	Verdadeiro se $arq for um arquivo texto.
`-B $arq`	Binary	Verdadeiro se $arq for um arquivo binário.

instruções `if` e `elsif` são executadas uma por vez, até que uma seja verdadeira ou até que a condição `else` seja alcançada. Quando atendida uma condição, o bloco correspondente é executado e os demais ignorados. As Figuras 4.35 e 4.36 mostram exemplos de uso dessas instruções.

```
#!/usr/bin/perl

$nome = "fulano";

if ($nome lt "ciclano") {
   print "Alfabeticamente, ’$nome’ vem antes de ’ciclano’.\n";
}
else {
   print "Alfabeticamente, ’ciclano’ vem antes de ’$nome’.\n";
}
```

Figura 4.35: Usando `if` e `else`.

unless

Talvez seja interessante não fazer nada quando a condição `if` for verdadeira, somente quando for falsa. O uso de um `if` vazio com um `else`, ou de uma negativa de `if` pode tornar o código confuso, segundo [Wall, et al. (2000)]. Para resolver esse problema usa-se

```
#!/usr/bin/perl

$resposta = "s";

if ($resposta eq "s") {
   print "Você␣escolheu␣'sim'.\n";
}
elsif ($resposta eq "n") {
   print "Você␣escolheu␣'não'.\n";
}
else {
   print "Você␣ainda␣está␣em␣dúvida?\n";
}
```

Figura 4.36: Usando `if`, `elsif` e `else`.

a instrução `unless` (a menos que). É bom frisar que não existe a instrução "elsunless". Na Figura 4.37 há um exemplo de uso do `unless`.

```
#!/usr/bin/perl

$cor = "vermelho";

unless ($cor eq "azul") {
    print "$cor␣não␣é␣a␣cor␣do␣céu.\n";
}
```

Figura 4.37: Usando o `unless`.

4.5.2 Estruturas de Repetição

while

A estrutura `while` executa um bloco enquanto uma condição for verdadeira. Quando esta condição se tornar falsa e for verificada novamente, o bloco não será mais executado. Se antes de executar o bloco a condição não for verdadeira, o bloco nunca chegará a ser executado. A Figura 4.38 mostra um exemplo de uso da instrução `while`.

until

A estrutura `until` é o contrário de `while`: executa um bloco enquanto uma condição for falsa. Quando esta condição se tornar verdadeira e for verificada novamente, o bloco não será mais executado. Se antes de executar o bloco a condição for verdadeira, o bloco nunca chegará a ser executado. Um exemplo do uso do `until` pode ser visto na Figura 4.39.

```
#!/usr/bin/perl

$i = 1;

while ($i <=10) {
    print "$i\n";
    $i++;
}
```

Figura 4.38: Usando o `while`.

```
#!/usr/bin/perl

$i = 1;

until ($i > 10) {
   print "$i\n";
   $i++;
}
```

Figura 4.39: Usando o `until`.

for

A instrução `for` utiliza três parâmetros: um estado inicial da variável de *loop*, uma condição e uma expressão para modificar a variável de *loop*. Quando o `for` inicializa, o estado inicial da variável de *loop* é definido e a condição é verificada. Se for verdadeira, o bloco é executado e a variável de *loop* modificada. Novamente a variável de *loop* é verificada. Se for verdadeira, repete-se o processo. Se for falsa, encerra-se o *loop*. A Figura 4.40 mostra um exemplo de uso do `for`.

```
#!/usr/bin/perl

@cor = ("azul", "vermelho", "verde", "amarelo", "laranja", "rosa");

for ($i=0 ; $i<=$#cor ; $i++) {
    print "$cor[$i] \n";
}
```

Figura 4.40: Usando o `for`.

foreach

Semelhante ao `for`, o `foreach` permite que um bloco seja executado para cada um dos valores do tipo *scalar* passados como parâmetro. Um valor do tipo *array* pode ser passado como parâmetro, pois a instrução `foreach` irá tratá-lo como uma lista de valores do tipo *scalar*. A variável de *loop* recebe cada elemento da lista individualmente, e o bloco é executado uma vez para cada elemento. Um exemplo do uso do `foreach` pode ser visto na Figura 4.41.

```
#!/usr/bin/perl

@cor = ("azul", "vermelho", "verde", "amarelo", "laranja", "rosa");

foreach $var (@cor) {
    print "$var␣\n";
}
```

Figura 4.41: Usando o `foreach`.

Instruções de Desvio – *next* e *last*

As instruções `next` e `last` podem modificar o fluxo original do *loop*. A instrução `next` permite pular para o final da iteração atual do *loop* e começar a próxima iteração. A instrução `last` permite pular para o final do bloco como se a condição de teste do *loop* tivesse retornado falsa. A Figura 4.42 mostra um exemplo de uso das instruções `next` e `last` e o seu resultado pode ser visto na Figura 4.43.

```
#!/usr/bin/perl

@cor = ("azul", "vermelho", "verde", "amarelo", "laranja", "rosa");

foreach $var (@cor) {
    if ($var eq "verde") {
       next;                    # retorna ao início e lê a próxima cor.
    }
    if ($var eq "laranja") {
       last;                    # sai do loop.
    }
    print "$var␣\n";
}
print "Fim!␣\n"
```

Figura 4.42: Usando o `next` e `last`.

```
[prompt]$ ./exemplo.pl
azul
vermelho
amarelo
Fim!
```

Figura 4.43: Resultado da execução do *script* da Figura 4.42.

4.6 *HANDLE* DE ARQUIVOS

Um *handle* de arquivos é apenas um nome que pode ser dado a um arquivo, dispositivo, soquete, ou pipe para simplificar a entrada e saída de dados de um programa em Perl. Primeiramente, é apresentado uma forma de ler dados do teclado e enviar resultados para a tela. A Figura 4.44 mostra um exemplo simples de uso de <STDIN> e <STDOUT>. O *handle* <STDIN> é o canal de entrada de dados padrão do *script*, enquanto que <STDOUT> é o canal de saída de dados padrão do *script*. Em sistemas *Unix-like*, os processos herdam entrada e saída do seu processo pai, que normalmente é um *shell*. No Linux, <STDIN> se refere à entrada padrão que é o teclado e <STDOUT> se refere à saída padrão que é a tela do monitor.

```
#!/usr/bin/perl

print STDOUT "Digite␣o␣seu␣nome:␣";
$nome = <STDIN>;
print STDOUT "Olá␣$nome,␣como␣vai?␣\n";
```

Figura 4.44: Usando o <STDIN> e <STDOUT>.

```
[prompt]$ ./exemplo.pl
Digite o seu nome: Herlon
Olá Herlon
, como vai?
[prompt]$
```

Figura 4.45: Resultado da execução do *script* da Figura 4.44.

Pode-se criar outras entradas e saídas de dados para os *scripts*. Essa é a principal finalidade dos *handles*. No exemplo da Figura 4.46 pode-se observar que o programa lê dados de um arquivo (`meutexto.txt`) e grava os resultados em outro arquivo (`outro.txt`). Para se inicializar um *handle*, usa-se a função `open()` e para encerrar o seu uso, usa-se a função `close()`. Na Tabela 4.8 pode-se visualizar algumas formas de se usar um *handle*.

```
#!/usr/bin/perl

open(ENTRADA, "meutexto.txt");
open(SAIDA, ">␣outro.txt");

while ($linha = <ENTRADA>) {
    print SAIDA "$linha";
}

close(SAIDA);
close(ENTRADA);
```

Figura 4.46: Lendo e gravando dados em arquivos.

Tabela 4.8: Definição de *handles*.

Handle	***Descrição***
`open(ARQ, 'arquivo')`	Lê do arquivo existente.
`open(ARQ, '< arquivo')`	Lê do arquivo existente.
`open(ARQ, '> arquivo')`	Cria o arquivo e grava nele.
`open(ARQ, '» arquivo')`	Acrescenta ao final do arquivo existente.
`open(ARQ, '| comando-pipe-saída')`	Prepara um filtro de saída.
`open(ARQ, 'comando-pipe-entrada |')`	Prepara um filtro de entrada.

O operador "< >" vazio lê linhas de todos os arquivos especificados na linha de comandos, ou de <STDIN>, se nenhum arquivo for especificado. Quando o <STDOUT> do sistema aponta para a tela, seu uso é redundante, pois a saída já iria para a tela do monitor de qualquer forma. Não há necessidade de escrever "<STDOUT>" quando o *script* enviar dados para a tela. No exemplo das Figuras 4.44 e 4.45 pode-se observar que foi recebido do teclado um sinal de mudança de linha. Isto acontece porque a operação de leitura de linha não remove automaticamente o caracter de nova linha "\n". Mas isso não é problema já que Perl possui a função `chomp()` que remove o caracter de escape "\n". Observa-se na Figura 4.47 como ficou o exemplo da Figura 4.44 agora reescrito com a função `chomp()` e o seu resultado é exibido na Figura4.48.

```
#!/usr/bin/perl

print STDOUT "Digite␣o␣seu␣nome:␣";
chomp($nome = <STDIN>);
print STDOUT "Olá␣$nome,␣como␣vai?␣\n";
```

Figura 4.47: Usando a função `chomp()`.

```
[prompt]$ ./exemplo.pl
Digite o seu nome: Herlon
Olá Herlon, como vai?
[prompt]$
```

Figura 4.48: Resultado da execução do *script* da Figura 4.47.

4.7 SUB-ROTINAS

As sub-rotinas, também chamadas de funções, são importantes para o reaproveitamento de código e por facilitar a legibilidade dos programas. Uma sub-rotina é definida pela declaração `sub` seguida do nome e do bloco de instruções. Esse bloco de instruções vem entre chaves "`{}`". As sub-rotinas não precisam ser declaradas antes do ponto onde são usadas. Para se executar uma sub-rotina coloca-se o marcador "`&`" antes de seu nome. Se não houver outra função ou variável com o mesmo nome da sub-rotina, e se esta for declarada antes de seu uso, este símbolo torna-se opcional. É uma boa prática de programação manter o marcador "`&`" antes do nome da sub-rotina.

Toda sub-rotina retorna um valor, que pode ser do tipo *scalar* ou não. Dentro da sub-rotina não é necessário indicar qual valor será retornado, pois ela retorna o valor da última expressão avaliada. Usando o comando `return` pode-se retornar outro valor diferente. A Figura 4.49 mostra um exemplo do uso de sub-rotinas.

```
#!/usr/bin/perl

$x = 3;
$y = 4;
$z = &soma($x,$y);                          # ou $z = soma $x, $y

print "A␣soma␣de␣$x␣e␣$y␣é␣$z.␣\n";

sub soma {
   $_[0] + $_[1];
}
```

Figura 4.49: Exemplo de uso de sub-rotinas.

Em Perl, a definição dos parâmetros que são passados para a sub-rotina é diferente das outras linguagens. Não se explicita ou se declara as variáveis que recebem os parâmetros passados para a sub-rotina. Em vez disso, Perl usa o *array* "@_", que contém uma lista com os argumentos passados para a função. É através desse *array* que as variáveis internas da sub-rotina recebem os valores transferidos.

4.7.1 Escopo de Variáveis

Blocos de instruções podem existir soltos no programa, dentro de estruturas de controle e também em sub-rotinas. Pode-se necessitar de variáveis que só façam efeito dentro desses blocos, as chamadas variáveis locais. Estas variáveis não interferem nas instruções fora do bloco a que pertencem. As variáveis locais podem ser criadas através do operador `my`. Essas variáveis deixarão de existir quando o bloco tiver terminado de ser executado.

Relacionada com o operador `my` está a diretiva "`use strict`". A diretiva força o interpretador Perl a aceitar apenas variáveis locais declaradas com o uso de `my`, ou seja, com essa diretiva fica obrigatória a declaração de variáveis através de `my`. Esse procedimento evita alguns tipos de erros de sintaxe, e a diretiva pode estar no início de um arquivo ou de um bloco de código. O exemplo apresentado na Figura 4.50 mostra o uso do operador `my` e da diretiva `use strict` e o resultado desse *script* pode ser visto na Figura 4.51.

```
#!/usr/bin/perl

use strict;

my $var = "global";

&imprime;
print "O␣valor␣de␣\$var␣fora␣da␣sub-rotina␣é:␣$var.␣\n";

sub imprime {
   my $var = "local";
   print "O␣valor␣de␣\$var␣dentro␣da␣sub-rotina␣é:␣$var.␣\n";
}
```

Figura 4.50: Usando a diretiva `use strict`.

```
[prompt]$ ./exemplo.pl
O valor de $var dentro da sub-rotina é: local.
O valor de $var fora da sub-rotina é: global.
```

Figura 4.51: Resultado da execução do *script* da Figura 4.50.

O *array* @ARGV é uma variável especial. Ela contém os argumentos que foram passados pela linha de comando. A Figura 4.52 mostra um exemplo de uso dessa variável e o resultado pode ser visto na Figura 4.53.

```
#!/usr/bin/perl

use strict;

my $origem = $ARGV[0];
my $destino = $ARGV[1];

print "Hoje␣viajarei␣de␣$origem␣até␣$destino.␣\n";
```

Figura 4.52: Usando o *array* @ARGV.

```
[prompt]$ ./exemplo.pl "Belo Horizonte" "São Paulo"
Hoje viajarei de Belo Horizonte até São Paulo.
```

Figura 4.53: Resultado da execução do *script* da Figura 4.52.

4.8 REFERÊNCIAS

Uma *referência* aponta para outra variável, ou seja, ela contém o endereço de outra variável. Normalmente é usada para passar parâmetros para uma função. Por exemplo, seja uma variável do tipo *array* de 1 MB de tamanho. Se esta variável for passada para uma sub-rotina, ela será copiada para a variável @_. Se for passada uma referência da variável e não a variável em si, economiza-se memória e ganha-se velocidade. A Figura 4.54 mostra um exemplo do emprego de referências. O sinal "\" (barra invertida) na frente da variável significa que está se extraindo sua referência. Pode-se referenciar *arrays*, *scalares*, *hashs*, *handles* e até mesmo *funções*.

A referência é um valor do tipo *scalar* ($) que pode, por exemplo, ser apontada para um *array* (@). Observa-se no exemplo da Figura 4.54 o procedimento para chamar de volta o valor apontado pela referência. Na frente do nome da referência (`$nome`) usa-se o símbolo de acordo com a variável referida. Por exemplo, um valor do tipo *array* seria chamado por `@$nome`, um *hash* por `%$nome` e um *scalar* por `$$nome`.

4.9 EXPRESSÕES REGULARES

Expressões Regulares, também conhecidas como *Regexp*, são utilizadas por muitos programas de pesquisa e manipulação de textos, como *Sed*, *Awk*, `grep`, etc. Sua principal

```
#!/usr/bin/perl

use strict;

my $a = "Estou␣aqui!";      # $a é uma variável scalar.
my $b = \$a;                # $b é uma referência para $a.

print "$$b␣\n";

my @vet_a = ("arroz", "feijão", "macarrão");
my $vet_b = \@vet_a;

print "@$vet_b␣\n";
```

Figura 4.54: Usando referências.

finalidade é determinar se uma *string* combina com um padrão específico ou não. Ela pode descrever um conjunto de *strings* sem ter que listar todas as *strings* do conjunto. Ou seja, ela tenta "casar" seu conteúdo com um pedaço de texto. Expressões regulares são um pouco diferentes em cada implementação, de forma que em Perl elas se diferem de Bash, Sed e Awk. Em [Jargas (2002)] e [Friedl (2002)] encontram-se boas fontes de informações sobre Expressões Regulares.

As expressões regulares são delimitadas por barras "`//`". Esta é uma forma abreviada de se usar o operador `m//`. Se for necessário usar a barra "`/`" como caracter normal dentro da expressão regular, ela deverá estar precedida da barra invertida "\" para perder seu significado especial de delimitador. No exemplo da Figura 4.55, procura-se pela *substring* "casa" dentro das *strings* exemplo. São fornecidas três *strings* de exemplo. Na primeira *string* não havia a *substring* "casa", portanto nada foi retornado. Na segunda *string* havia a *substring* "casa", portanto o operador `//` retornou a parte da *string* exemplo que se encaixava na regra definida. Na terceira *string*, foi retornada a palavra "casados" pois a *substring* da regra se encaixou nesta palavra. O operador de associação de padrões "`=~`" está dizendo ao interpretador Perl para procurar uma combinação da expressão regular "casa" dentro das *strings* exemplo.

Se nas *strings* exemplo houvesse a palavra "Casa", com "C" maiúsculo, o operador `m//` não retornaria essa ocorrência. Este operador diferencia maiúsculas de minúsculas. Se não for de interesse esta diferenciação, pode-se usar o modificador "`i`", após a segunda barra, que faz com que maiúsculas e minúsculas deixem de ser diferenciadas.

Existe também o operador `s///` que além de determinar se uma *string* combina com um padrão definido, ele também faz substituições no texto em caso afirmativo. No exemplo da Figura 4.56, o comando `$texto = s/Windows/Linux/` substitui a palavra "Janelas" por "Linux" na frase, na primeira ocorrência. Se a frase possuir mais de uma ocorrência,

```
#!/usr/bin/perl

use strict;

my $texto1 = "A␣moradia␣em␣apartamentos␣tem␣sido␣a␣solução␣em␣grandes␣cidades.";
my $texto2 = "No␣interior,␣casa␣é␣uma␣boa␣opção.";
my $texto3 = "O␣jogo␣será␣entre␣casados␣e␣solteiros.";

if ($texto1 =~ /casa/) {
   print "Encontado␣em␣\$texto1.␣\n";
}

if ($texto2 =~ /casa/) {
   print "Encontado␣em␣\$texto2.␣\n";
}

if ($texto3 =~ /casa/) {
   print "Encontado␣em␣\$texto3.␣\n";
}
```

Figura 4.55: Exemplo de uso de expressões regulares.

e se for desejável a substituição de todas, é necessário adicionar o modificador “g” após a última barra.

```
#!/usr/bin/perl

use strict;

my $texto = "Estou␣usando␣Janelas,␣porque␣gosto␣muito␣de␣Janelas.";

$texto =~ s/Janelas/Linux/;

print "$texto␣\n";
```

Figura 4.56: Exemplo de uso do `s///`.

Expressões regulares são utilizadas, normalmente, para encontrar ocorrências mais complexas do que simples palavras. Por exemplo, encontrar um endereço ip no formato “`w.x.y.z`” dentro de uma *string*. Para realizar essas procuras mais complexas há uma série de caracteres com significado especial que serão usados numa expressão regular. A Tabela 3.8 da Seção 3.5 do Capítulo 3 mostra vários desses caracteres, que também têm o mesmo significado em Perl, com seus respectivos significados e exemplos.

Algumas classes de caracteres são muito comuns e bastante utilizadas. Para simplificar, pode-se fazer uso de códigos especiais. A Tabela 4.9 mostra esses códigos especiais

com seus respectivos significados. Pode-se usar também letras maiúsculas para indicar qualquer caracter exceto os que forem da própria classe.

Tabela 4.9: Códigos especiais de classes de caracteres.

Código	***Descrição***
\d	Classe dos dígitos, [0-9].
\w	Classe de letras, dígitos e *underscore*, [A-Za-z0-9_].
\s	Espaço em branco.

O "^" fora de colchetes ([]) indica que a regra deverá ocorrer no início da linha. O "$" indica que a regra deverá ocorrer no final da *string*. O caracter de nova linha "\n" poderá aparecer em qualquer lugar no interior da *string*, que não alterará o resultado.

Pode-se utilizar parênteses para indicar agrupamentos de elementos e, também, para se lembrar de partes que já foram combinadas. Um par de parênteses em torno de uma parte de uma expressão regular faz com que aquilo que foi combinado por essa parte seja lembrado para uso posterior. Isso não altera as regras da expressão regular. O modo como se refere à parte lembrada depende de onde se quer fazer isso. Dentro da mesma expressão regular são usados os termos \1, \2, \3, ..., para o primeiro, segundo, terceiro, ..., par de parênteses encontrados, respectivamente. Se a referência à parte lembrada for feita fora da expressão regular, usa-se os termos $1, $2, $3, ..., no lugar de \1, \2, \3, ... Um exemplo pode ser visto na Figura 4.57

```
#!/usr/bin/perl

$frase = "o␣doce␣perguntou␣pro␣doce␣qual␣era␣o␣doce␣mais␣doce␣que␣o␣doce␣de
batata-doce.";

if ($frase =~ /^(o) (doce) perguntou pro \2 qual era \1 \2 mais \2 que \1 \2
de batata-\2./) {
    print "Primeiro␣parêntese:␣$1␣\n";
    print "Segundo␣parêntese:␣$2␣\n";
}
```

Figura 4.57: Exemplo de referências posteriores.

Existem, também, as variáveis especiais $‘, $& e $’, que representam, respectivamente, o que vem antes, a parte encontrada e o que vem depois na string analisada.

4.10 EXEMPLO FINAL – UM ANALISADOR DE LOGS

Esta seção encerra o conteúdo de Perl implementando um exemplo mais completo, abrangendo os conceitos vistos até aqui. Este exemplo é um caso real muito comum no dia-a-dia de um administrador de sistemas. Será desenvolvido um *script* para análise do arquivo de *log* de um servidor *Web*. Esse exemplo é baseado em [Dias (2000)].

O formato padrão do conteúdo de um arquivo de *log* do servidor *Web* Apache[6] é mostrado na Figura 4.58. O significado de cada campo é mostrado na Tabela 4.10. Na Figura 4.59 há um exemplo real de uma linha qualquer encontrada num arquivo de *log*.

```
endereço    ident    user    [data]    "request"    status    bytes
```

Figura 4.58: Formato padrão de um arquivo de *log* do servidor *Web*.

Tabela 4.10: Significado dos campos no arquivo de *log* do servidor *Web*.

Campo	***Descrição***
`endereço`	Endereço IP da máquina do cliente.
`ident`	Resposta do *ident* no cliente.
`user`	Nome do usuário, caso tenha se autenticado.
`data`	Data e hora do acesso.
`request`	Requisição enviada pelo cliente.
`status`	*Status* respondido pelo servidor.
`bytes`	Quantidade de *bytes* transferido.

```
200.141.51.23 - - [23/Jan/2005:16:14:32 +0000] "GET /index.htm HTTP/1.1" 200 686
```

Figura 4.59: Exemplo do conteúdo de um arquivo de *log* do servidor *Web*.

Do arquivo de log, necessita-se obter as seguintes informações:

- as páginas mais acessadas;
- o endereço ip dos usuários que mais acessam o servidor;
- a quantidade de *bytes* transferidos por página;

[6] Mais detalhes podem ser obtidos em `http://httpd.apache.org/docs-project/`.

- a quantidade de *bytes* transferidos por clientes;
- a quantidade total de *bytes* transferidos.

Nota-se que as informações desejadas se encontram nos campos *endereço*, *request* e *bytes*. Nos interessa apenas as requisições aceitas. O servidor *Web* registra o valor "`200`" no campo *status* quando a requisição foi aceita. As outras informações serão ignoradas.

A idéia principal do programa consiste em:

- ler todas as linhas do arquivo de *log*, uma a uma;
- verificar se essas linhas estão no formato padrão do arquivo de *log*;
- extrair os campos *endereço*, *request*, *status* e *bytes*;
- trabalhar com as linhas que possuem *status* igual a `200` (aceito);
- contabilizar o número de requisições de cada endereço ip;
- contabilizar o número de requisições de cada página;
- contabilizar a quantidade de *bytes* solicitados por cada endereço ip;
- contabilizar a quantidade de *bytes* fornecidos por cada página;
- contabilizar o total de *bytes* transferidos pelo servidor;
- colocar em ordem decrescente as páginas mais acessadas;
- colocar em ordem decrescente os clientes que mais acessam o servidor;
- exibir resultados.

O *script* será desenvolvido por etapas. A primeira é apresentada na Figura 4.60 e consiste em definir a localização do interpretador Perl, as diretivas utilizadas e declarar as variáveis.

```
!/usr/bin/perl -w

use strict;

# declaração das variáveis
my ($ip, $full_request, $status, $bytes);
my ($request, $total, $nao_200, $erros);
my (%contagem_ip, %contagem_request, %bytes_ip, %bytes_request);

$erros = 0;
```

Figura 4.60: Primeira etapa do exemplo "Analisador de *logs*".

A segunda etapa consiste em ler linha por linha do arquivo de log e extrair as informações necessárias. A Figura 4.61 mostra mais detalhes. Geralmente o servidor *Web* gera

vários arquivos de *log*. Talvez seja mais interessante que os nomes desses arquivos sejam passados na linha de comando, como parâmetros do programa, em vez de inseri-los dentro do código. Isto permite uma maior flexibilidade. Para isso é utilizada a estrutura `while (<>) {...}` que faz com que todos os arquivos passados na linha de comando sejam abertos para leitura. A outra alternativa, um pouco mais trabalhosa, seria o uso do comando `open` para a criação de um *handle*.

```
while (<>){                                  # loop para ler cada linha
  if ( /^(\S+)\s+\S+\s+\S+\s+\[.*\]\s+"(.*)"\s+(\S+)\s+(\S+)$/ ){
    $ip      = $1;
    $full_request = $2;
    $status    = $3;
    $bytes    = $4;
    if ($3 == 200){
      #Extrai a URL:
      $full_request =~ /\S+\s+(.*)\s+\S+/;
      $request = $1;
      if ($bytes eq "-"){                    # Garante que o valor é numérico
        $bytes = 0;
      }
      $contagem_ip{$ip}++;                   # Contagem de acessos por ip.
      $contagem_request{$request}++;         # Contagem de acessos por página.
      $bytes_ip{$ip} += $bytes;              # Contagem de bytes transf. por ip.
      $bytes_request{$request} += $bytes;    # Contagem de bytes transf. por pág.
      $total += $bytes;                      # Contagem do total de bytes transf.
    }else{
      $nao_200++;                            # Cont. pág. c/ status dif. de 200.
    }
  }else{                                     # Contagem das linhas que não "casam"
  $erros++;                                  # com a Expressão Regular.
  }
}                                            # Fim do loop while.
```

Figura 4.61: Segunda etapa do exemplo “Analisador de *logs*”.

Cada linha extraída é comparada com a expressão regular chamada no exemplo de “principal” para verificar se atende ao formato padrão do arquivo de *log* do servidor *Web*. As linhas que não se casarem serão contabilizadas como erro. Observa-se na Figura 4.61 que essa expressão regular possui quatro pares de parênteses, que extraem, respectivamente, os campos *endereço*, *request*, *status* e *bytes*, que são atribuídos às suas respectivas variáveis.

Nos interessa apenas as linhas com o campo *status* igual a `200` que significa que a requisição foi aceita e enviada. As linhas que possuírem o campo *status* diferente de `200` serão contabilizadas como “requisições não aceitas”.

O campo *request* traz, além da URL solicitada, o método de envio e o protocolo utilizado. É necessário extrair desse campo apenas a URL. Para isso, utiliza-se uma outra expressão regular, chamada no exemplo de "secundária". A URL extraída é armazenada em outra variável.

O servidor *web* coloca um símbolo "`-`" no campo bytes caso o cliente aborte a requisição. Se for encontrada esta situação, a quantidade de bytes transmitida, nesse caso, será considerada igual a zero.

De posse de todas as informações necessárias, pode-se fazer a contagem das páginas acessadas, da quantidade de *bytes* transferida e dos clientes que acessaram o servidor. Essas contagens serão feitas com variáveis do tipo *hash*, onde as chaves serão ou a URL da página solicitada ou o ip do cliente, e os valores correspondentes a essas chaves serão a contagem propriamente dita.

A terceira e última etapa é a ordenação e exibição dos resultados. Foi construída uma sub-rotina para executar esta tarefa. A Figura 4.62 mostra mais detalhes. Os dados são passados para a sub-rotina através de referências, ordenados numericamente e exibidos os dez maiores valores. A função `sort()` só ordena em ordem lexicográfica e não pode ser usada diretamente para ordenar números. É preciso entender como a função `sort()` trabalha: ela compara dois valores e retorna, internamente para seu próprio algoritmo, `-1` se o primeiro for menor, `0` se for igual, e `1` se for maior. Pode-se construir um bloco de instruções para a função `sort()`[7], onde através do operador <=> pode-se comparar números de forma "numérica", podendo retornar os valores `-1`, `0` e `1` para o algoritmo interno da função `sort()`. As variáveis `$a` e `$b` são variáveis padrões temporárias[8] utilizadas pela função `sort()`. A função `reverse()` foi usada para colocar a ordenação em ordem decrescente, uma vez que `sort()` ordena em ordem crescente. Feita essa ordenação, pega-se os dez primeiros valores (`[0..9]`) para serem exibidos.

O código completo deste exemplo pode ser obtido unindo-se os códigos das Figuras 4.60, 4.61 e 4.62. A Figura 4.63 mostra o resultado de sua execução.

[7] Há mais informações sobre a construção de blocos para a função `sort()` nas páginas de manual (`man perl`) ou em `http://perldoc.perldrunks.org/functions/sort.html`.

[8] É uma boa prática não usar variáveis definidas pelo usuário com nomes `$a` e `$b`.

```
#As 10 páginas (arquivos) mais acessadas
print "Páginas mais acessadas:\n\n#\tacessos\tbytes\tPáginas\n";
ordena_e_mostra(\%contagem_request, \%bytes_request);

#Os 10 clientes (usuários) que mais acessam
print "\nClientes que mais acessam:\n\n#\tacessos\tbytes\tCliente\n";
ordena_e_mostra(\%contagem_ip, \%bytes_ip);

#Resultados finais
print "Erros:  $erros\n";
print "Requisições não aceitas: $nao_200\n";
print "Total bytes:  $total\n";

#Função que ordena e mostra os dados
sub ordena_e_mostra{
  my ($ref_a, $ref_b) = @_;
  my $count = 0;
  foreach my $item ((reverse sort {$$ref_a {$a} <=> $$ref_a {$b}} keys \\
  %$ref_a) [0..9]){
    $count++;                    # adiciona 1 ao $count
    print "$count\t";            # Mostra os dados
    print "$$ref_a{$item}\t";    # Item principal
    print "$$ref_b{$item}\t";    # Item secundário
    print "$item\n";             # Nome do item
  }
}
```

Figura 4.62: Terceira etapa do exemplo “Analisador de *logs*”.

```
[prompt]$ ./analisador.pl access.log
Páginas mais acessadas:

#       acessos bytes   Páginas
1       12      6480    /cgi-bin/monitux/sobre.cgi
2       7       2534    /cgi-bin/monitux/home.cgi
3       3       16581   /cgi-bin/wxis.exe?IsisScript=phl7/003.xis&cipar=phl...
4       3       1917    /cgi-bin/monitux/usuarios.cgi
5       2       2280    /cgi-bin/monitux/servicos.cgi
6       2       1850    /cgi-bin/monitux/configuracao.cgi
7       2       2384    /cgi-bin/wxis.exe?IsisScript=phl7/004.xis&cipar=phl...
8       2       288     /cgi-bin/monitux/frame_off.cgi
9       1       1654    /cgi-bin/monitux/screens.cgi
10      1       522     /phl7/ingles.jpg

Clientes que mais acessam:

#       acessos bytes   Cliente
1       72      401860  127.0.0.1
2       15       27639  192.168.254.3
3       13       33290  192.168.254.21
4       12       18398  192.168.254.15
5       9        12897  192.168.254.8
6       7        21654  192.168.254.2
7       6        31065  192.168.254.11
8       3        17541  192.168.254.6
9       1         5419  192.168.254.4
10      1         6431  192.168.254.5

Erros:  0
Requisições não aceitas: 5
Total bytes: 701660
```

Figura 4.63: Resultado da execução do exemplo “Analisador de *logs*”.

5

FERRAMENTAS DE DESENVOLVIMENTO

5.1 INTRODUÇÃO

Compilar um programa num sistema diferente do qual ele foi desenvolvido pode ser uma tarefa complexa. Existe uma grande quantidade de variáveis que precisam ser analisadas e que afetam a maneira como o programa precisa ser compilado. Isso é especialmente importante em sistemas como o Linux, onde dificilmente existiram dois computadores configurados exatamente da mesma forma.

As pessoas que desenvolvem programas para sistemas da família Unix foram criando ferramentas ao longo dos anos para simplificar o processo de compilação de um programa, levando em consideração coisas como:

- a existência ou não de determinada biblioteca de funções instalada para uma uma determinada linguagem;
- a versão dessas bibliotecas;
- o sistema de arquivos utilizado;
- as configurações de segurança no sistema de arquivo;
- a organização da árvore de diretórios de um determinado computador.

O desenvolvimento dessas ferramentas acabou sendo absorvido pelo projeto GNU, o que levou ao desenvolvimento de ferramentas como *autoconf*, *automake*, *autoheader* e *libtool* (doravante chamadas de *autotools*) que buscam simplificar o processo de compilação de um programa, facilitando assim o desenvolvimento de programas portáveis. Essas ferramentas possuem a seguinte funcionalidade:

***Autoconf*:** realiza testes no sistema onde é executado, procurando características que afetem a forma como um determinador programa precisa ser compilado. Ele proporciona maneiras de adaptar o código fonte ao sistema onde ele será compilado, através de scripts (usando *bash*, por exemplo).

***Automake*:** cria arquivos *makefile* que são regras para controlar a execução do compilador, simplficando a interação entre o programador e o compilador.

***Autoheader*:** cria arquivos de cabeçalho (arquivos .h) que são usados como forma de adaptar o código fonte de um programa às características de uma plataforma. Arquivos .h são exclusivos das linguagens de programação C e C++, porém essas linguagens são usadas no desenvolvimento da quase totalidade dos programas compiláveis em sitemas da família Unix.

***Libtool*:** oferece uma interface entre programador e compilador/ligador, com a finalidade de simplificar as questões de ligação (tanto estática quanto dinâmica) do programa executável.

Essas ferramentas evitam que o desenvolvedor de um programa precise conhecer detalhes das diversas plataformas onde o programa poderá ser compilado. Elas acrescentam um passo a mais no processo de transformação de código fonte em código executável: a configuração.

5.2 O PROGRAMA *MAKE*

Um primeiro passo para entender o funcionamente de ferramentas de desenvolvimento é entender o programa *make* e o formato de seus arquivos de controle, chamados *makefile*.

O nome *make* vem do inglês, e quando verbo significa “fazer”, “construir”, “criar”. Quem não gostaria de, após desenvolver o código de um determinado programa, simplesmente dizer para o computador “crie o programa *x*”? A idéia do programa *make* é justamente essa: alguém chegaria para o computador e diria “make editor”, e automaticamente, o programa `editor` seria criado.

Para que alguém possa ter essa facilidade toda, outra pessoa precisa cuidar de todo o trabalho. O usuário de um programa (distribuído na forma de código fonte) não deveria precisar saber quais programas usar na compilação, quais opções passar para esses programas, etc. Este trabalho deveria ficar todo nas mãos de quem desenvolve o programa e portanto cabe aos desenvolvedores entender o formato *makefile*, e criar um arquivo no qual serão colocadas as informações necessárias para que o programa *make* faça a compilação.

O próprio desenvolvedor também ganha com o uso do *make*. Durante a evolução do ciclo de desenvolvimento de um programa, certamente serão realizadas várias compilações. Se o desenvolvedor “ensina” o computador sobre como o programa dele deve ser compilado, vai gastar mais tempo uma vez, para poupar tempo várias vezes depois.

Na verdade, o *make* não serve somente para compilar programas, ele é usado em qualquer situação onde é necessário transformar arquivos num formato mais voltado para o tratamento por humanos em arquivos num formato mais voltado para o tratamento por máquinas. Exemplos de uso do *make* incluem transformação de arquivos texto em bases de

dados e transformação de descrições de documentos (como código LaTeX) em documentos propriamente ditos (como arquivos PDF).

Um arquivo *makefile* é um conjunto de regras que dizem ao *make* como transformar alguns arquivos em outros. Por exemplo: suponha que um programa está divido em quatro arquivos: `arq1.c`, `arq2.c`, `arq1.h` e `arq2.h`. Suponha ainda que para transformá-los num arquivo executável `editor`, é necessário compilar `arq1.c` e `arq1.h` em `arq1.o`, compilar `arq2.c`, `arq1.h` e `arq2.h` em `arq2.o` e finalmente ligar o código objeto de `arq1.o` e `arq2.o` criando o código executável do arquivo `editor`. Podemos identificar aqui três regras a serem descritas no arquivo *makefile*:

1. como criar `arq1.o` a partir de `arq1.c` e `arq1.h`;
2. como criar `arq2.o` a partir de `arq2.c`, `arq1.h` e `arq2.h`;
3. como criar `editor` a partir de `arq1.o` e `arq2.o`.

Descritas essas regras, quando o `make` for acionado para criar `editor`, ele vai notar que precisa do `arq1.o` e `arq2.o`. Sabendo disso, ele procura a regra que indica como `arq1.o` é criado, notando que para isso precisa de `arq1.c` e `arq1.h`.

Não existem regras dizendo como criar `arq1.c` e `arq1.h`, portanto eles precisam existir antes da execução do `make`[1]. Caso não existam, o `make` aborta sua execução informando que houve erro. Caso existam, ele tenta verificar se `arq1.c` ou `arq1.h` são mais recentes do que `arq1.o`. Se algum deles for, siginifica que `arq1.o` precisa ser refeito de acordo com as instruções contidas no arquivo *makefile*. Por outro lado, se `arq1.o` for mais recente que os dois arquivos necessários para criá-lo, não há porque perder tempo refazendo-o. Essa mesma verificação é feita com `arq2.o` e finalmente feita com `editor`.

```
alvo ...: prerequisito ...
        comando
        ...
```

Figura 5.1: Formato geral de uma regra num arquivo *makefile*

Uma regra tem o formato geral mostrado na Figura 5.1. Ela começa com uma linha que contém a descrição de um ou mais alvos (separados por espaços). Geralmente cada regra tem um único alvo, que é o nome de um arquivo que o `make` deve ser capaz de criar. Nessa mesma linha, separados dos alvos pelo caracter dois pontos (':'), vêm os pré-requisitos. Eles são geralmente nomes de arquivos que são necessários para se criar os alvos indicados, quando isso for necessário.

[1] Na verdade, existem várias regras implícitas que normalmente são usadas sem que estejam descritas no arquivo *makefile*. Para maiores informações sobre regras implícitas, consulte a documentação do `make`.

Comandos são definições de como se usar os pré-requisitos para se gerar os alvos. Eles devem aparecem em linhas iniciadas por um caracter de tabulação (representados por setas na figura)[2] e são escritos da mesma forma que seriam escritos num interpretador de linha de comando.

Cada regra serve para dizer como determinar se um alvo precisa ser refeito e para indicar qual procedimento deve ser usado. Um alvo será refeito se ele não existir e for necessário para criar o alvo que o usuário pediu, ou então se algum de seus pré-requisitos for mais recente que ele.

```
editor: arq1.o arq2.o
        gcc -o editor arq1.o arq2.o

arq1.o: arq1.c arq1.h
        gcc -c arq1.c

arq2.o: arq2.c arq1.h arq2.h
        gcc -c arq2.c
```

Figura 5.2: Exemplo de arquivo no formato *makefile*

A Figura 5.2 apresenta um arquivo no formato *makefile* que contém as regras sobre a criação do arquivo `editor`, comentado anteriormente. Os comandos são simplesmente a chamada do compilador com seus devidos parâmetros para gerar os arquivos desejados.

Um arquivo no formato *makefile* pode conter outras coisas além de regras. Isto será abordado a seguir. Ele deve ser salvo preferencialmente no diretório onde estão os arquivos envolvidos em sua descrição, facilitando a digitação dos nomes. Seu próprio nome não precisa seguir nenhuma regra e deve ser passado para o `make` via opção '`-f`', como em `make -f meuarquivo editor`, onde o `make` é instruido a criar `editor`, usando o arquivo de regras `meuarquivo`. Quando não se usa a opção '`-f`', o `make` procura por alguns arquivos padrão. A versão GNU do *make* (geralmente a versão disponível nos sitemas Linux), procura por arquivos chamados `GNUmakefile`, `makefile` e `Makefile`, nessa ordem. O nome mais comum é `Makefile` pois a inicial maiúscula faz com que o arquivo apareça no início da listagem de um diretório na maioria dos sistemas. Criado o arquivo `Makefile`, conforme a Figura 5.2, podemos criar o arquivo `editor` com o comando `make editor` conforme a Figura 5.3.

Suponha que durante a execução de `editor`, foi identificada uma falha no programa. Essa falha foi corrigida através de uma alteração no arquivo `arq2.h`. Ao recompilar o programa, o *make* detecta que precisa ser gerado um novo `arq2.o` e consequentemente

[2] Muitas pessoas aprendendo a usar o `make` cometem o erro de deixar espaços antes dos comandos, até mesmo porque muitos editores de código fonte substituem automaticamente tabulações por espaços. Nesse caso, o `make` vai acusar erro de sintaxe.

```
[prompt]$ make editor
gcc -c arq1.c
gcc -c arq2.c
gcc -o editor arq1.o arq2.o
[prompt]$ ls
arq1.c  arq1.h  arq1.o  arq2.c  arq2.h  arq2.o  editor  Makefile
[prompt]$ ./editor
```

Figura 5.3: Exemplo de uso do `make`

um novo `editor`, mas não recompila `arq1.c` para gerar um novo `arq1.o`, conforme pode ser visto na Figura 5.4. Isso pode significar uma boa economia de tempo, dependendo do tamanho dos programas envolvidos.

```
[prompt]$ make editor
gcc -c arq2.c
gcc -o editor arq1.o arq2.o
[prompt]$
```

Figura 5.4: Somente os arquivos afetados pela alteração de outro são refeitos

Quando se omite qual o alvo deve ser usado, o `make` usa o primeiro alvo que existe no arquivo. Costuma-se então, deixar o alvo que faz tudo que é necessário para criação do programa na primeira posição, e os alvos intermediários depois. Existem nomes tradicionais para alvos que, se usados, exigirão que se escreva menos documentação a respeito de como compilar o programa. Os alvos tracionais mais comuns são (mais ou menos em ordem de popularidade):

1. `all` compila tudo o que for possível, possivelmente incluindo programas auxiliares usados na depuração do programa principal - costuma ser o primeiro alvo;
2. `clean` apaga todos os arquivos gerados durante a compilação dos programas, deixando apenas o código fonte;
3. `install` copia executáveis, arquivos de dados e documentação para diretórios globais como `/usr/local/bin`, `/usr/share`, etc., alterando permissões conforme necessário para que o programa passe a estar disponível para todos os usuários do computador - para fazer isso, é preciso ter privilégios de administrador;
4. `uninstall` apaga todos os arquivos instalados pelo alvo `install` - também precisa de privilégios de administrador.

Além de regras, arquivos *makefile* usam variáveis para guardar texto (geralmente nomes de arquivos ou opções a serem passadas para um programa). Isso pode facilitar a

manutenção e a portabilidade do arquivo *makefile*. Por exemplo: na Figura 5.2, foi usado o compilador `gcc`, mas em alguns computadores, o compilador C usado não é o da GNU e se chama `cc`[3]. Para compilar o editor numa máquina assim, seria necessário trocar o nome do compilador nos três lugares em que ele aparece (e poderiam ser 30 lugares). Para facilitar isso, é interessante guardar o nome do compilador numa variável e usar seu valor para se referir ao compilador, conforme mostrado na Figura 5.5. Se o nome do compilador precisar ser alterado, uma única mudança será suficiente.

```
# Altere aqui para indicar o nome do seu compilador C
compilador = gcc

# Regras
editor: arq1.o arq2.o
        $(compilador) -o editor arq1.o arq2.o

arq1.o: arq1.c arq1.h
        $(compilador) -c arq1.c

arq2.o: arq2.c arq1.h arq2.h
        $(compilador) -c arq2.c

all: editor

clean:
        rm -f editor *.o
```

Figura 5.5: Uso de variáveis e comentários num arquivo *makefile*

Existem muitas variáveis pré-definidas que podem e geralmente são usadas num arquivo *makefile*. As variáveis pré-definidas mais usadas são[4]:

- `CC`: programa usado para compilar arquivos escritos em linguagem C - seu valor padrão é `cc`;
- `CXX`: programa usado para compilar arquivos escritos em linguagem C++ - seu valor padrão é `g++`;
- `CPP`: programa usado para fazer pré-processamente em arquivos C - seu valor padrão é `$(CC) -E`;
- `PC`: programa usado para compilar arquivos escritos em linguagem Pascal - seu valor padrão é `pc`;
- `RM`: programa usado para apagar arquivos - seu valor padrão é `rm -f`.

[3] Em computadores com Linux e o *gcc* instalado, `cc` é um ponteiro para `gcc`

[4] Variáveis válidas para o `make` da GNU, não necessáriamente pré-definidas em outras implementações.

Algumas variáveis contém nomes estranhos de um único caracter. Nestes casos, não é necessário colocar os parênteses em volta do nome da variável (após do sinal de dólar) para se obter o seu valor. Elas são usadas na parte dos comandos de uma regra para indicar valores úteis:

- @: nome do alvo sendo seguido - útil para regras baseadas em alvos múltiplos como aquelas baseadas em padrões;
- <: nome do primeiro pré-requisito - útil para regras baseadas em padrões;
- ?: nomes de todos os pré-requisitos (separados por espaços) que são mais novos que o alvo;
- ^: nomes de todos os pré-requisitos;
- *: parte variável de um padrão usado como alvo.

Os nomes dessas váriaveis, seguidos de 'D' ou 'F' indicam respectivamente o nome do diretório e do arquivo (sem a parte do diretório) dos valores apresentados anteriormente. Existem também variáveis pré-definidas cujo valor padrão é *string* vazia, usados para passar parâmetros para os programas mais usados:

- `CFLAGS`: parâmetros do compilador C;
- `CXXFLAGS`: parâmetros do compilador C++;
- `LIBS`: bibliotecas a serem usadas na etapa de ligação (parâmetros '-l' e '-L' do *linker*;
- `LDFLAGS`: parâmetros extras a serem usados pelo compilação na etapa de ligação de código objeto (como '-s' para retirar símbolos do código executável).

Você já deve estar pensando que é possível automatizar grande parte da criação de arquivos *makefile*. Não seria muito difícil fazer um programa que abre os vários arquivos com código fonte de um programa, procura instruções como `#include ¨lista.h¨` e cria um arquivo *makefile* baseado no próprio código fonte. As ferramentas *autotools* podem automatizar esse processo e num nível bem maior do que simplesmente consultar qual arquivo inclui qual[5].

5.3 *AUTOTOOLS*

A grande maiorias dos usuários de Linux e sistemas operacionais semelhantes já instalou (ou tentou instalar) um programa distribuído na forma de código fonte copiado da Internet. Na maioria da vezes, a documentação do programa diz para executar um *script* chamado `configure`, depois executar o programa `make` e depois executar `make install`.

[5]Isso não é necessariamente uma vantagem, já que algumas vezes seria desejável algo mais simples.

O `configure` é um *script* Bash que realiza diversos testes no sistema, procurando programas, bibliotecas e outras coisas necessárias à compilação do programa que se quer instalar. Na pior das hipóteses o `configure` avisa sobre problemas que deverão aparecer durante a compilação *antes* de se tentar compilar o programa, afinal a compilação é um processo que pode tomar um bom tempo, e nem sempre as mensagens de erro do compilador são apropriadas para quem quer apenas instalar o programa. Na melhor da hipóteses, o `configure` detecta problemas e os corrige automaticamente, adaptando o processo de compilação à máquina sendo utilizada. Ele cria o arquivo *makefile* baseado no resultado dos testes, além de permitir que o usuário compilando o programa personalize diversas coisas, como o diretorio onde o programa deverá ser instalado ou quais opções do programa serão ativadas.

Arquivos como o *script* `configure` são longos e difícieis de serem criados manualmente. O programa `autoconf` serve justamente para criar tal *script*, utilizando-se dos programas `m4` e `aclocal`. O programa `m4` é um "processador de macros", ou seja, ele transforma arquivos texto que usam mnemônicos como `FUNC(parâmetro1, parâmetro2, ...)` em arquivos texto com uma sintaxe mais sofisticada como `template <class parâmetro1> parâmetro1 clNohLista<parâmetro1>::Funcao(parâmetro2,...)`. O `m4` é muito usado para facilitar a escrita de arquivos de configuração, como o do servidor `sendmail`, por exemplo. O programa `aclocal` será apresentado posteriormente.

Antes de criar um arquivo `configure`, é preciso descrever num arquivo denominado `configure.in` quais testes são necessários para garantir que seu programa pode ser compilado sem nenhum problema. Essa descrição é feita através de uma série de "macros" que o processador `m4` transforma em comandos no arquivo `configure`. Como o *script* `configure` vai gerar um arquivo `Makefile`, é preciso também descrever as dependências de compilação e outras coisas que são importantes para a criação do `Makefile`.

Aprender a usar o *autoconf* envolve dois itens fundamentais: conhecer as dezenas de macros que descrevem testes[6] e saber o que precisa ser testado para resolver problemas de portabilidade num programa específico. Esse segundo item é, sem sombra de dúvida, a parte mais difícil de se usar o *autoconf*. Para os objetivos deste texto vamos supor que qualquer novo projeto será desenvolvido em C++ (e não em C), usando uma distribuição Linux (e não outras variações de Unix) baseada em pacotes de *software* (como RPM) onde a preocupação quase sempre é saber se os pacotes necessários à compilação estão instalados.

A maior parte da documentação dos *autotools* é voltada para dúvidas como "posso usar a função `time` de C para saber a hora corrente?" ou "será que é possível converter *strings* em números usando `atol`?", de forma que dúvidas mais avançadas (e mais inco-

[6]Para isto, a melhor coisa a fazer é ler a documentação do *autoconf*. O comando `info autoconf` será suficiente na maioria dos casos.

muns) como estas podem ser mais facilmente resolvidas com uma simples busca na Internet (em especial, recomenda-se a leitura de [Bergo (2001)]). Para uma referência mais completa sobre as ferramentas *autotools*, recomenda-se a leitura de [Vaughan, et al. (2001)].

```
AC_INIT(main.cpp)

dnl Qual e' o compilador C++?
AC_PROG_CXX
AC_LANG_CPLUSPLUS

AC_PROG_MAKE_SET

dnl AC_HEADER_STDC
dnl AC_CHECK_FUNC(atol,,AC_MSG_ERROR(oops! no atol ?!?))

AC_CHECK_LIB(ncurses,main,,AC_MSG_ERROR(instale a biblioteca 'ncurses'!))

VERSION="0.0.1"
AC_SUBST(VERSION)

dnl ler Makefile.in e escrever Makefile
AC_OUTPUT(Makefile)
```

Figura 5.6: Arquivo `configure.in` para checar a existência da biblioteca `ncurses`.

Na Figura 5.6, pode-se notar uma série de macros em destaque, a finalidade de cada uma delas é:

- `AC_INIT`: serve para inicialização de checagem de erros no *autoconf*; deve ser a primeira macro no arquivo e deve receber o nome de um arquivo no diretório onde está o código fonte do programa (costuma-se usar o arquivo principal);
- `dnl`: indica comentário (em qualquer arquivo processado pelo *m4*), vale até o final da linha;
- `AC_PROG_CXX`: procura qual o compilador de C++ e verifica nele a existência de determinadas características como, por exemplo, se o compilador é capaz de gerar código com símbolos para depuração;
- `AC_LANG_CPLUSPLUS`: indica que o programa usa a linguagem C++;
- `AC_PROG_MAKE_SET`: verifica se o `make` tem a variável *MAKE* pré-definida; isso é importante porque muitas vezes os comandos de uma regra chamam o `make` para realizar tarefas dentro de sub-diretórios;
- `AC_CHECK_LIB`: verifica se é possível compilar o programa usando determinada biblioteca - recebe o nome de uma função da biblioteca para usar como teste (`main` é usado quando não se deseja testar uma função em particular);

- AC_SUBST: indica que o valor de uma determinada variável deve ser passado adiante na criação do arquivo Makefile;
- AC_OUTPUT: determina a execução do *autoconf*, gerando os arquivos necessários - deve ser a última macro no final do arquivo.

A variável VERSION foi usada aqui com a finalidade de mostrar como o valor de uma variável pode ser passado do configure.in até chegar no código fonte, alterando o comportamento do programa. Não há aqui a intenção de propor uma forma de controle de versões de programa usando esse sistema.

O objetivo principal do exemplo do arquivo mostrado na Figura 5.6 é criar um *script* configure que verifica se é possível compilar um programa com a biblioteca *ncurses* (sem preocupações com a versão da biblioteca), além de aproveitar para associar valores válidos às variáveis tradicionais (ver página 130) a serem usadas no Makefile.

```
CXX = @CXX@
VERSAO = @VERSION@
CXXFLAGS = @CXXFLAGS@
LIBS = @LIBS@
LDFLAGS = @LDFLAGS@

alocurses: main.o
        $(CXX) $(LIBS) $(LDFLAGS) $< -o $@

main.o: main.cpp
        $(CXX) $(CXXFLAGS) -DVERSAO=\"$(VERSAO)\" -c $<

clean:
        $(RM) alocurses core *.o

distclean:
        $(RM) alocurses config.* *.o Makefile

all: alocurses
```

Figura 5.7: Modelo usado na criação do Makefile pelo *script* configure

Além do arquivo configure.in, é necessário ter um modelo de *makefile* para a criação do Makefile durante a execução do configure, esse arquivo é o Makefile.in. Um exemplo pode ser visto na Figura 5.7 para um programa que usa a biblioteca "ncurses". Ele começa passando valores para variáveis que serão usadas no Makefile como em CXX = @CXX@ onde a variável CXX a ser usada no Makefile recebe o valor obtido (automaticamente em AC_PROG_CXX) pelo configure.

O resto do Makefile.in é examente como um arquivo *makefile*. Note como o valor de VERSION descrito em configure.in é passado para VERSAO em Makefile, que por

sua vez, passa o valor para o código fonte através da opção `-D` do compilador, criando o símbolo VERSAO de forma equivalente à declaração de `#define VERSAO ¨0.0.1¨` diretamente no código fonte.

Tendo esses dois arquivos, basta executar `autoconf` no diretório em questão para que seja criado o script `configure`. Executando o `configure` numa máquina onde "ncurses" e "ncurses-devel" não estejam ambas instaladas, deve-se ver algo semelhante à Figura 5.8.

```
[prompt]$ ./configure
creating cache ./config.cache
checking for c++... c++
checking whether the C++ compiler (c++  ) works... yes
checking whether the C++ compiler (c++  ) is a cross-compiler... no
checking whether we are using GNU C++... yes
checking whether c++ accepts -g... yes
checking whether make sets ${MAKE}... yes
checking for main in -lncurses... no
configure: error: instale a biblioteca 'ncurses'!
[prompt]$
```

Figura 5.8: Exemplo de verificação executada pelo `configure`

Caso as bibliotecas estejam instaladas, o `Makefile` será criado e o resultado da execução do *make* será aquele mostrado na Figura 5.9. No caso o compilador usado foi c++ ao invés de g++, que é normalmente o preferido, mas usado pelo *autoconf* somente quando não é possível usar o primeiro.

```
[prompt]$ make
c++ -g -O2 -DVERSAO=\"0.0.1\" -c main.cpp
c++ -lncurses   main.o -o alocurses
[prompt]$
```

Figura 5.9: Compilação do programa que usa a biblioteca "ncurses"

A forma como a *string* que identifica a versão do programa foi passada para o programa funciona, mas é pouco prática quando se quer passar uns 40 valores. Uma forma mais prática de se fazer isso seria criar um arquivo de cabeçalho (arquivo com extensão '.h') e se colocar esses valores no arquivo. O programa `autoheader` faz exatamente isso.

Para isso, vamos colocar no `configure.in` a macro `AC_CONFIG_HEADER` que faz com que os valores coletados pelo `configure` sejam colocados num arquivo (usalmente chamado `config.h`). Colocando `AC_CONFIG_HEADER(config.h)` logo após `AC_INIT`, e executando `autoheader`, será criado um arquivo `config.h.in` como apresentado na

Figura 5.10. Ele contém símbolos que permitem saber se existe ou não determinada biblioteca, função, programa, etc. Esses símbolos, por sua vez, podem ser usados para alterar a forma como o programa é compilado, através do uso de `#ifdefs` ou `#ifs`. Logicamente, o código fonte precisa incluir o código do `config.h` para ter acesso aos resultados dos testes.

```
/* config.h.in. Generated automatically from configure.in by autoheader. */

/* Define if you have the ncurses library (-lncurses).  */
#undef HAVE_LIBNCURSES
```

Figura 5.10: Arquivo gerado pelo *autoheader*

Depois de executado o `configure`, será criado um arquivo `config.h`, conforme mostrado na Figura 5.11. Neste caso há somente um símbolo, já que foi descrito apenas um teste no `configure.in`. Se fosse usado um teste como `AC_CHECK_FUNCS(atol atoi strtod)`, que verifica se é possível usar as funções `atol`, `atoi` e `strtod`, haveriam três outros símbolos com os resultados do teste.

```
/* config.h.  Generated automatically by configure.  */
/* config.h.in. Generated automatically from configure.in by autoheader. */

/* Define if you have the ncurses library (-lncurses).  */
#define HAVE_LIBNCURSES 1
```

Figura 5.11: Arquivo `config.h` gerado pelo `configure`

Ao invés de criar um arquivo `Makefile.in` diretamente, como mostrado anteriormente, costuma-se deixar esta tarefa para o programa `automake`. Para isso é necessário criar um arquivo `Makefile.am` que será usado para criar o `Makefile.in`. A criação do `Makefile.am` é muito parecida com a criação do arquivo `configure.in`, onde são usadas macros com o objetivo de reduzir a quantidade de texto que o programador deve escrever. Novamente, recomenda-se a leitura da documentação do *automake* para saber quais macros existem. O *automake* cria arquivos `Makefile` um pouco mais sofisticados, criando automaticamente alvos como `install`, `uninstall`, `dist`, `distclean`, etc. que, de outra forma, precisariam ser criados manualmente. Por outro lado, para programas simples, o `Makefile` criado irá parecer sofisticado demais, com uma grande quantidade de variáveis, testes e alvos que acabam dificultando sua leitura.

Antes de criar um `Makefile.am` que será usado para criar um `Makefile.in` equivalente ao antigo (Figura 5.7), deve-se modificar o `configure.in`, colocando nele instruções para criação do novo `Makefile.in`. O arquivo `Makefile.am` pode ser visto na Figura 5.12. As alterações são:

- a inicialização do *automake*, com a macro `AM_INIT_AUTOMAKE`[7], logo após a macro `AC_INIT`, passando o nome do programa (que será usado para nomear o principal arquivo executável) e a sua versão (usada para dar valor à variável `VERSION`);
- a substituição de `AC_CONFIG_HEADER` por `AM_CONFIG_HEADER`;
- a eliminação da parte que trata da variável `VERSION`, que mais tarde era passada como `VERSAO`, no momento da chamada do compilador, pois o projeto usa *autoheader* e a variável `VERSION` criada com `AM_INIT_AUTOMAKE` vai automaticamente para o `config.h`.

```
AC_INIT(main.cpp)
AM_INIT_AUTOMAKE(alocurses,0.0.1)
AM_CONFIG_HEADER(config.h)

dnl Qual e' o compilador C++?
AC_PROG_CXX
AC_LANG_CPLUSPLUS

AC_PROG_MAKE_SET

AC_CHECK_LIB(ncurses,main,,AC_MSG_ERROR(instale a biblioteca 'ncurses'!))

AC_OUTPUT(Makefile)
```

Figura 5.12: Exemplo de `configure.in` para uso do *automake*

O próximo passo é criar o arquivo `Makefile.am`, conforme a Figura 5.13. A macro `AUTOMAKE_OPTIONS` é usada para definir alguns padrões para o funcionamento global do *automake*. Normalmente se usa opção `gnu` para criar programas distribuídos sob a licença GPL, mas o exemplo apresenta a opção `foreign` que deixa o *automake* menos exigente a respeito do que o pacote de distribuição deve conter. Isso será comentado a seguir.

```
AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS = alocurses
alocurses_SOURCES = main.cpp
```

Figura 5.13: Exemplo de `Makefile.am` para criação do `Makefile.in`

A macro `bin_PROGRAMS` serve para indicar quais são os programas (arquivos executáveis) que devem ser criados além de dizer que, quando instalados, os programas devem ir

[7] Note que as macros do *automake* começam com "AM" ao contrário das macros do *autoconf*, que começam com "AC".

para o diretório especificado pela variável `bindir` (normalmente `/usr/local/bin`). Programas que devem ser usados somente pelo super-usuário podem ser indentificados pela macro `sbin_PROGRAMS`. A grande variedade de macros do *automake* permite fornecer rapidamente toda a informação necessária para a construção dos vários alvos no arquivo `Makefile`. Consulte a documentação do *automake* para encontrar macros para as necessidades do seu programa específico.

A macro `alocurses_SOURCES` serve para indicar quais são os arquivos que devem ser compilados para criação do programa (alvo) `alocurses`. A parte mínuscula das macros, como você deve estar percebendo, é variável e relativa ao programa especificado. A parte escrita em maiúsculas é fixa e descrita na documentação do *automake*.

Depois de criado o arquivo `Makefile.am`, é possível criar os outros arquivos de configuração automaticamente. As regras do *automake*, colocadas no `configure.in` não são conhecidas pelo *autoconf*, mas podem ser criadas, quase da mesma forma que um usuário pode criar suas próprias macros e adicioná-las às outras com o programa `aclocal`. A diferença é que ao invés de criar um arquivo `acinclude.m4` que seria usado pelo *aclocal* para criar o `aclocal.m4` (que conteria as novas macros), executar `aclocal` sem criar nenhuma descrição é suficiente. O *aclocal* encontra as definições do *automake* no diretório `/usr/share/aclocal`, onde existem também macros para facilitar o uso de bibliotecas específicas como “GTK”, “qthreads” dentre outras que podem estar instaladas na sua máquina.

Para criar os arquivos de configuração, deve-se então executar os programas `aclocal`, `autoheader`, `autoconf` e `automake`, nesta ordem, como no comando `aclocal && autoheader && autoconf && automake -a`. A opção ’-a’ ordena que o *automake* crie[8] automaticamente alguns arquivos cuja existência aumenta a portabilidade de uma distribuição (usados em alvos como `install`), a saber:

- `install-sh`: *script* que serve para copiar arquivos alterando suas permissões, controlado por variáveis de nomes pré-determinados;
- `mkinstalldirs`: *script* usado para criar diretórios caso não existam;
- `missing`: *script* usado para checar a existência de arquivos ou diretórios alertando o usuário a respeito de problemas de instalação com mensagens padronizadas.

O uso da opção `gnu` na macro `AUTOMAKE_OPTIONS`, exigiria também a existência dos arquivos `INSTALL`, `COPYING`, `NEWS`, `README`, `AUTHORS` e `ChangeLog`, com declarações sobre licença de uso e documentação do programa de uma maneira geral. Os dois primeiros podem ser copiados de modelos do *automake* (pela opção ’-a’), mas os quatros últimos precisam realmente ser escritos.

[8] Na verdade os arquivos são copiados de `/usr/share/automake` ou, mais precisamente, são feitos *links* simbólicos que irão virar arquivos quando o pacote do programa for criado pelo uso do alvo `dist`.

6

RPMS

6.1 INTRODUÇÃO

A sigla RPM significa *Red Hat Package Manager* (Gerenciador *Red Hat* de Pacotes)[1], visto que o RPM foi criado pela empresa *Red Hat* para ser usado na distribuição Linux que leva seu nome, entretanto como o RPM é distribuído sob licença GPL, várias outras distribuições adotam o RPM. Um "pacote" é um conjunto de arquivos (executáveis, dados e documentação) que formam um aplicativo (ou parte de um aplicativo) como por exemplo um editor de textos. O nome "rpm" é usado para indicar o formato dos arquivos que contém pacotes RPM e é também o nome do programa que gerencia esses pacotes. Como exemplo de formatos semelhantes ao RPM, podemos citar o "MSI" da MicroSoft e o DEBIAN (arquivos '.deb') da distribuição *Debian*.

Gerenciar pacotes significa instalar, atualizar e desinstalar programas. A tarefa é mais complicada do que simplesmente copiar e apagar arquivos pois muitos programas dependem de outros (ou dependendem de bibliotecas) para funcionar. Uma característica de destaque do formato RPM é capacidade de tratar tanto programas distribuídos na forma de código fonte (em arquivos chamados de *source RPMs*) como aqueles distribuídos na forma binário (arquivos executáveis já compilados).

O objetivo deste texto é apresentar conceitos sobre RPM em quantidade suficiente para se conseguir criar um pacote RPM a partir do código fonte de um programa. Para uma referência mais completa, recomenda-se a leitura de [Bailey (1998)] e [Barnes (1999)]. Entre os motivos principais para uso do formato RPM estão:

- maior portabilidade do código (o formato RPM já foi convertido, entre outros, para o Solaris e o Windows);
- facilidade de administração dos programas e bibliotecas instaladas (utilizando-se arquivos RPMs bem construídos é praticamente impossível ter conflito de bibliotecas);

[1] Alguns preferem *RPM Package Manager* - que é uma boa maneira de tornar a sigla recursiva, como tantas outras no mundo Unix

- facilidade de atualização de pacotes[2] e
- verificação de integridade antes e depois da instalação.

Observe que mesmo distribuições Linux não baseadas no formato RPM, como o *Slackware*, acabam por criar aplicativos que tentam proporcionar essa funcionalidade.

6.2 CRIANDO PACOTES RPM

No Capítulo 5, foi apresentado o uso de ferramentas de compilação e distribuição num exemplo bastante simples, chamado de “alocurses”, que é um programa que usa a biblioteca “ncurses”. Vamos continuar o seu desenvolvimento de maneira a criar um pacote RPM.

O primeiro passo é criar um arquivo *spec* que contém a descrição do programa, instruções sobre como ele deve ser compilado e uma lista de arquivos que serão instalados. O nome desse arquivo, por convenção, deve ser algo como *nome-versão-lançamento.spec*[3] que, no exemplo apresentado na Figura 6.1, seria `alocurses-0.0.1-1.spec`.

Na primeira parte do arquivo, conhecida como preâmbulo, pode-se ver uma seção com definições de valores para variáveis auto-explicativas. Existem outras variáveis, mas as apresentadas aqui são suficientes para pacotes pequenos. Os pares “variável: valor” do preâmbulo não podem ocupar mais de uma linha. A variável `source` é usada na hora da compilação do programa, e por isso a última parte da URL (ou o valor todo, mesmo que não seja uma URL) deve ser um nome verdadeiro. A seguir, aparece uma seção de descrição do pacote, demarcada pelo símbolo `%description`. É essa descrição que é apresentada pelo programa `rpm` quando se pede informações sobre um pacote.

Depois da seção de descrição, aparecem, respectivamente as seções de preparação, compilação e instalação do programa. Na seção de preparação coloca-se comandos para descompactar e deixar os arquivos num estágio próprio para a compilação. No exemplo, usou-se a macro `%setup` que automaticamente apaga arquivos de possíveis compilações anteriores e descompactar o arquivos presentes no *tar.gz*. Após isso é executado o *script* `configure` que deverá gerar o arquivo `Makefile`. É comum o uso desta seção para aplicação de *patches* ao código fonte ou execução de scripts que tornam possível a compilação de programas de terceiros sem que haja necessidade de alterar seu código fonte diretamente.

Na seção de compilação, basta executar o `make`, afinal o arquivo *makefile* foi preparado desta forma. Para instalar para chamar o alvo `install`. A seção demarcada por

[2] Por pacote entende-se qualquer conjunto de arquivos, como: programas, bibliotecas e documentação, entre outros.

[3] O termo lançamento vem do inglês *release* e é usado para indicar nova distribuição de versão que já havia sido distribuída antes, possivelmente usando opções diferentes no processo de compilação.

```
Summary: Exemplo de criacao de pacotes RPM
Name: alocurses
Version: 0.0.1
Release: 1
Copyright: dominio publico
Group: Applications/Inutilities
Source: ftp://algum.lugar.net/alocurses-0.0.1.tar.gz
URL: http://www.nenhum.lugar.net/inutilidades/alocurses/
Packager: Bruno Schneider <ninguem@nenhum.lugar.net>
Requires: ncurses >= 3.0

%description
Este e' um programa que usa a biblioteca 'ncurses'. Ele foi usado como
exemplo de criacao de pacotes RPM. Instale se estiver aprendendo a criar
pacotes RPM.

%prep
%setup
./configure

%build
make

%install
make install

%clean
# fazer nada

%files
/usr/local/bin/alocurses

%changelog
* Sun Oct 06 2002 Bruno Schneider <ninguem@nenhum.lugar.net>
- Criada a primeira versao do spec.
```

Figura 6.1: Exemplo de arquivo *spec*

`%clean` serve para guardar comandos que apaguem arquivos temporários fora do diretório normal do programa, o que não foi necessário no exemplo.

Em seguida aparece a seção demarcada por `%files`, que contém uma lista de todos os arquivos que farão parte do pacote, que por sua vez, são os arquivos instalados pelo alvo `install`. Não existe uma maneira fácil de se fazer essa lista automaticamente, recomenda-se que ela seja feita manualmente e com **muita** atenção. Por fim, aparece a seção demarcada por `%changelog` onde são documentadas modificações feitas no programa.

Para seja possível criar o pacote RPM, é necessário certificar-se que os seguintes pré-requisitos estão satisfeitos:

- o pacote `rpm-build` deve estar instalado;
- você pode conseguir permissão de super-usuário para o processo de criação do pacote (na verdade isso não é realmente necessário, mas criar pacotes como usuário comum é um processo trabalhoso - é bom saber de ante-mão que será executado o alvo `install`, de forma que seu *makefile* deve proporcionar formas de fazê-lo como usuário comum - verifique a documentação do RPM se for realmente necessário);
- existe uma cópia do seu programa no formato “tar.gz” no diretório utilizado pelo *rpm-build* (`/usr/src/redhat/SOURCES` nas distribuições Red Hat).

Caso tudo esteja certo, execute como super-usuário o comando `rpmbuild -ba --clean alocurses-0.0.1-1.spec`. Isto fará com que sejam criados dois arquivos: um com o sufixo ‘.src.rpm’ e outro com o sufixo ‘.i386.rpm’. O primeiro contém o arquivo *spec* e o arquivo *.tar.gz* enquanto que o segundo contém o *spec* e os arquivos (compilados) listados na seção `%files` do *spec*. Durante o processo de criação dos arquivos RPM, é feita uma checagem automática para verificar quais bibliotecas dinâmicas são usadas pelo programa e essas informações são colocadas no pacote compilado.

6.3 SEGURANÇA

Apesar de ser muito popular entre a maioria dos usuários de computadores, a instalação de programas executáveis é uma pratica insegura, muito mal vista por alguns defensores dos programas livres. Entre os principais argumentos contra ela, está o fato de que os programas poderiam ser alterados, de forma a conter código maligno e seria muito difícil detectar alguma alteração.

Na prática, a instalação de código fonte, compilado na máquina do usuário apresenta riscos semelhantes, afinal existe uma grande distância entre a disponibilização de código fonte e a leitura do mesmo. O formato RPM procura resolver o problema da segurança com a utilização de criptografia. Através de criptografia, os pacotes RPM podem receber uma *assinatura digital*, que identifica a procedência do pacote RPM, além de garantir que o mesmo não foi alterado por terceiros.

Para utilizar o tipo de criptografia utilizada nos pacotes RPM, é necessário ter instalado um programa compatível com o programa *Pretty Good Privacy* (PGP[4]). Numa instalação Linux, o mais recomendado é o programa *GNU Privacy Guard* (GPG[5]). Utilizar ferramentas

[4]PGP: `http://www.pgpi.org/`.

[5]GPG ou GnuPG: `http://www.gnupg.org/`.

de criptografia está fora do escopo deste texto. Os exemplos descritos a seguir supõe que o usuário tem um par de chaves (pública e privada) para assinar pacotes RPM, além das chaves públicas dos distribuidores de pacotes RPM cuja autenticidade será testada[6]. Para maiores informações sobre o GPG, consulte a documentação do mesmo, disponível em sua página.

Pode-se adicionar uma assinatura digital a pacotes RPM durante a criação do pacote ou depois da mesma. Um pacote RPM pode ser assinado por várias pessoas ou organizações ao mesmo tempo. Quando um pacote é assinado, seus usuários não são obrigados a saber trabalhar com criptografia a não ser que queiram checar a validade da assinatura.

Para adicionar uma assinatura digital a pacotes RPM, é necessário primeiro especificar qual o programa que será usado e quem irá fazer as assinaturas. Essa configuração do programa `rpm` pode ser global, através do arquivo `/etc/rpm/macros` ou local (para um determinado usuário) no diretório `~/.rpmmacros`.

Seja qual for o arquivo de configuração usado, deve-se idenficar qual o programa a ser utilizado pelo identificador `%_signature` (atualmente, os valores possíveis são `pgp` ou `gpg`). Deve-se determinar a identidade de quem vai assinar os pacotes pela macro `%_gpg_name` (a identificação deve ser exatamente igual à usado no programa de criptografia). Se a identificação não pertencer ao chaveiro digital do usuário que executar o `rpm`, pode-se definir a localização do diretório com as chaves pelo identificador `%_gpg_path`. No caso de uso do programa `pgp`, todos símbolos devem ser alterados de forma a conter “pgp” no lugar de “gpg”.

```
%_signature gpg
%_gpg_name Projeto Ginux <ninguem@nenhum.lugar.net>
```

Figura 6.2: Arquivo de configuração do `rpm` para uso de criptografia

A Figura 6.2 mostra um exemplo de arquivo de configuração que indica que o `gpg` será usado para tratar a criptografia e que os pacotes que forem assinados receberão a assinatura identificada por “Projeto Ginux <ninguem@nenhum.lugar.net>”.

Depois de configurado é facil incluir a assinatura digital num pacote. Para fazer isso durante o processo de criação do pacote, basta incluir a opção `--sign` durante a criação (opção `-ba`) do pacote. Nesse caso, a primeira coisa que o `rpm` fará é pedir a senha para a assinatura. Caso ela esteja correta, os arquivos são compilados e instalados para criação

[6] A chave pública usada para verificar a integridade de pacotes distribuídos pela RedHat pode ser encontrada no arquivo `/usr/share/doc/redhat-release-?.?/RPM-GPG-KEY` (os pontos de interrogação são o número da versão da distribuição). A maioria das distribuições Linux têm um arquivo com sua chave pública nos CDs de distribuição.

do pacote como ocorre normalmente. No final da criação do pacote, o `rpm` deverá exibir mensagens como as apresentadas na Figura 6.3.

```
Generating signature: 1005
Generating signature using GPG.
Wrote: /usr/src/redhat/SRPMS/alocurses-0.0.1-1.src.rpm
Generating signature: 1005
Generating signature using GPG.
```

Figura 6.3: Mensagens relativas à assinatura de um pacote RPM sendo criado

Se o pacote ja existe, é possivel adicionar uma assinatura, com a opção `--resign` como em `rpm --resign` *`nome-do-pacote.rpm`* ou com a opção `--addsign` que substitui a assinatura anterior.

Para verificar se um pacote está corretamente assinado, indicando que sua integridade é confirmada pela pessoa ou entidade possidora da assinatura, basta usar a opção `-K`. A Figura 6.4 mostra um caso de assinatura correta (para o pacote `alocurses-0.0.1-1.i386.rpm`) e um caso de assinatura incorreta (para o pacote `nss_ldap-186-1.src.rpm`).

```
[prompt]$ rpm -K alocurses-0.0.1-1.i386.rpm
alocurses-0.0.1-1.i386.rpm: md5 gpg OK
[prompt]$ rpm -K nss_ldap-186-1.src.rpm
nss_ldap-186-1.src.rpm: md5 (GPG) NOT OK (MISSING KEYS: GPG#F9651D5A)
[prompt]$
```

Figura 6.4: Verificação da integridade de pacotes RPM

O programa `rpm` escreve ao lado do nome do pacote o que está sendo testado e quais testes falharam. No primeiro caso, os testes “md5” e “gpg” resultaram em sucesso. O teste “md5” é um *checksum* (um tipo de código para verificação de erros) disponível em qualquer pacote RPM que serve para verificar se o arquivo não está corrompido. Esse teste não é suficiente para saber se o arquivo não foi deliberadamente alterado por alguém mal intencionado. O teste “gpg” é efetivamente o teste da assinatura digital. Se um teste falha, como no caso do pacote `nss_ldap-186-1.src.rpm`, o nome do teste aparece entre parênteses. Ainda neste exemplo, há ao lado uma explicação de que o teste “gpg” falhou por falta da chave pública de quem assinou o pacote.

7

AGENDAMENTO DE TAREFAS

7.1 INTRODUÇÃO

O Linux possui um recurso muito útil para o administrador de sistemas que é a possibilidade de se agendar uma tarefa para ser executada num determinado dia e horário, sem a necessidade do administrador estar presente no momento. Com isso, pode-se programar tarefas administrativas para serem realizadas em horários pré-definidos, sem intervenção humana. Tarefas como realização de *backups* periódicos, rotação automática de *logs*, remoção de *links* simbólicos quebrados, atualização de pacotes, entre tantas outras, podem ser programadas para serem executadas em horários e/ou dias em que o sistema estiver com uma carga menor de serviços. Neste capítulo são apresentados os agendadores de tarefas Cron e At. Informações mais completas podem ser obtidas nas páginas do respectivos manuais.

7.2 USO DO CRON

7.2.1 Características Gerais

O Cron permite agendar tarefas para serem executadas num exato momento, podendo ser especificados o mês, dia do mês e/ou da semana, hora e minuto. Essas tarefas podem ser rotineiras, repetidas várias vezes num intervalo pré-determinado. O Cron consiste de um *daemon* chamado `crond`, que é inicializado junto com o sistema, e de arquivos de configuração chamados de `crontab` (*cron table* ou tabela cron).

Através dos arquivos `cron.allow` e `cron.deny`, o administrador de sistemas pode determinar quais usuários podem ou não usar os serviços do Cron. O usuários que estiverem incluídos no arquivo `cron.allow` terão permissão para usar o Cron, enquanto que os usuários com o nome em `cron.deny` não terão acesso ao Cron. Se o arquivo `cron.allow` não existir, todos os usuários terão permissão de usar o Cron, exceto os que estiverem listados em `cron.deny`. Se nenhum dos dois existir, todos os usuários poderão

usar o Cron. A localização desses dois arquivos pode variar em função de cada distribuição, mas geralmente ficam em `/etc/cron.d`.

As tarefas agendadas pelos usuários são configuradas em arquivos `crontab` que ficam localizados em `/var/spool/cron/`. Cada usuário terá um arquivo individual com o seu próprio nome. Assim, o usuário `fulano` terá o arquivo `/var/spool/cron/fulano` com as suas configurações.

7.2.2 Formato do Arquivo Crontab

Os arquivos `crontab` têm o seu formato apresentado na Figura 7.1. Os campos `minuto`, `hora`, `dia`, `mês` e `dia-da-semana` informam quando executar o `comando`. Seus valores possíveis são mostrados na Tabela 7.1, e um exemplo é apresentado na Figura 7.2. Nesse exemplo, o arquivo `aviso.sh` será executado em todos os meses, de segunda-feira à sexta-feira, às 07:30 horas da manhã.

```
minuto hora dia mês dia-da-semana comando
```

Figura 7.1: Formato das linhas do arquivo `crontab`.

Tabela 7.1: Valores possíveis nos campos `crontab`.

Campo	***Valores***
`minuto`	0 a 59
`hora`	0 a 23
`dia`	1 a 31
`mês`	1 a 12
`dia-da-semana`	0 a 6 (0 corresponde ao domingo)

```
30 7  *  *  1-5   /usr/local/bin/aviso.sh
```

Figura 7.2: Exemplo de um arquivo `crontab`.

Nota-se no exemplo da Figura 7.2 que, além dos números mostrados na Tabela 7.1, estão presentes os símbolos asterisco (`*`) e hífen (`-`). O asterisco representa todos os valores possíveis e o hífen define um intervalo de valores. Há também a barra (`/`) que pode definir valores uniformemente espaçados, e a vírgula (`,`) que define uma lista de valores. A Figura 7.3 mostra um exemplo em que uma tarefa é executada de 10 em 10 minutos, durante os sábados e domingos.

```
*/6   *  *  *  0,6   /usr/local/bin/vassoura.sh
```

Figura 7.3: Utilizando intervalos num arquivo `crontab`.

7.2.3 Criando um Arquivo Crontab

Para se criar um arquivo `crontab` usa-se o comando de mesmo nome, `crontab`, que possui as opções mostradas na Tabela 7.2. Se o usuário ainda não tiver um arquivo `crontab`, este deverá ser criado a partir de um editor de textos qualquer. No exemplo da Figura 7.4 é mostrado o arquivo de texto `teste_cron` com instruções para ser usado com o `crontab`. A Figura 7.5 mostra a criação do arquivo `crontab`. Esse arquivo `crontab` será criado no diretório `/var/spool/cron` com o mesmo nome do usuário que o criou. Para editá-lo, usa-se apenas `crontab -e` e automaticamente é aberto o arquivo `crontab` do usuário. Para apenas listar o conteúdo, usa-se `crontab -l`, e para removê-lo, usa-se `crontab -r`.

Tabela 7.2: Opções do comando `crontab`.

Opção	***Descrição***
`-e`	Edita o arquivo `crontab` atual do usuário.
`-l`	Lista o conteúdo do arquivo `crontab` do usuário.
`-r`	Remove o arquivo `crontab` do usuário.

```
30 5  *  *  1  rm -rf /tmp/*
*/30  *  *  *  1-5   /usr/local/bin/monitora.sh
15 7,17  *  *  1,3,5 /home/fulano/lembranca.sh
```

Figura 7.4: Tarefas para o `crontab`.

```
[prompt]$ crontab teste_cron
```

Figura 7.5: Criando um arquivo `crontab`.

O usuário `root` tem acesso a todos os arquivos `crontab` dos usuários e, também a um arquivo `crontab` para agendamento de tarefas do sistema: `/etc/crontab`. Esse arquivo possui um campo a mais onde o administrador do sistema pode especificar o nome do usuário que executará o comando. A Figura 7.6 mostra um exemplo desse arquivo. No início do arquivo, o Cron define algumas variáveis para a sua execução. O comando

`run-parts` é responsável pela execução de todos os *scripts* que estiverem dentro dos diretórios correspondentes. Cada diretório é executado com uma periodicidade diferente. A Tabela 7.3 mostra a periodicidade de cada diretório.

```
# /etc/crontab: system-wide crontab

SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/

# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

Figura 7.6: Arquivo `/etc/crontab`.

Tabela 7.3: Periodicidade dos diretórios contidos no arquivo `/etc/crontab`.

Diretório	***Periodicidade***
`/etc/cron.hourly`	De hora em hora.
`/etc/cron.daily`	Diariamente.
`/etc/cron.weekly`	Semanalmente.
`/etc/cron.monthly`	Mensalmente.

O uso desses diretórios é muito simples, bastando apenas que se coloque o *script* desejado, com permissão de execução, dentro do respectivo diretório. Por exemplo, um *script* que será executado diariamente será colocado no diretório `/etc/cron.daily`, e um que será executado de hora em hora será colocado no diretório `/etc/cron.hourly`. Isso facilita o trabalho do administrador de sistemas, que pode separar os *scripts* em diretórios conforme sua periodicidade.

7.3 USO DO AT

7.3.1 Características Gerais

Ao contrário do Cron, o At não pode ser usado para agendar tarefas que se repetem periodicamente, ou seja, deve ser usado somente para tarefas que serão executadas

apenas uma vez. Uma característica muito útil do At é a possibilidade de execução de programas que passaram do horário de sua execução, devido ao computador estar desligado no momento ou falta de energia elétrica.

Assim como o Cron, o At possui dois arquivos para permitirem ou negarem seu uso: `/etc/at.allow` e `/etc/at.deny`, que funcionam da mesma forma que os arquivos `cron.allow` e `cron.deny`, respectivamente (Seção 7.2.1).

A sintaxe do comando `at` é mostrada na Figura 7.7, onde `opções` são as mostradas na Tabela 7.4. O campo `tempo` pode ser preenchido com os valores da Tabela 7.5.

```
at [opções] tempo
```

Figura 7.7: Sintaxe do comando `at`.

Tabela 7.4: Opções do comando `at`.

Opção	***Descrição***
`-c`	Mostra as tarefas registradas.
`-d`	Remove uma tarefa específica, o mesmo que `atrm`.
`-f`	Lê as tarefas a partir de um arquivo.
`-l`	Lista as tarefas agendadas pelo usuário, o mesmo que `atq`.
`-m`	Envia um e-mail ao usuário quando a tarefa for finalizada.
`-v`	Informa a hora em que uma tarefa será executada.

7.3.2 Usando o At

O `at` aceita comandos de três formas. A primeira e mais utilizada é através da entrada padrão, quando o usuário digita o comando `at`, suas opções e o tempo, e se abre um segundo shell para que se digite a tarefa a ser agendada e finaliza com um sinal de *EOF* (`<Ctrl+D>`). Veja um exemplo na Figura 7.8.

A segunda forma é ler a tarefa a ser agendada de um arquivo, através da opção `-f`, e a terceira, é através de um redirecionamento através de um *pipe* (`|`). A Figura 7.9 mostra exemplos de uso do `at`.

Tabela 7.5: Valores permitidos para o campo `tempo`.

Campo tempo	*Descrição*
`hh:mm [modificadores]`	Assume-se, por padrão, um relógio de 24 horas. Se usado os modificadores `am` ou `pm`, usa-se um relógio de 12 horas.
`midnight, noon, teatime` e `now`	Significam *meia-noite*, *meio-dia*, *16:00hs* e *agora*, respectivamente. Substituem o formato numérico. O `now` é usado em conjunto como sinal "+" seguido de um número e uma das palavras-chave: `minute`, `hour`, `day`, `week`, `month`, ou `year`.
`mês dia[,ano]`	O `mês` é um dos doze meses escrito por completo ou com as três primeiras letras. O `dia` é um número entre `1` a `31`, e o `ano` é formado de quatro dígitos.
`dia`	Representa o dia da semana escrito por inteiro ou com as três primeiras letras.
`today, tomorrow`	Indica o dia atual ou o dia seguinte, respectivamente.

```
[prompt]$ at 12:02 today
at> cd /home/herlon; mkdir teste_at
at> <EOT>    <================  ATENÇÂO: Isso não foi digitado, e sim <CTRL+D>
warning: commands will be executed using /bin/sh
job 8 at 2005-03-05 12:02
[prompt]$
```

Figura 7.8: Usando o `at` através da entrada padrão.

```
[prompt]$ at -f tarefa_at.txt 5 pm Saturday
warning: commands will be executed using /bin/sh
job 10 at 2005-03-06 17:00
[prompt]$ echo "cd /home/herlon; mkdir teste_at_2" | at 12:15 today
warning: commands will be executed using /bin/sh
job 11 at 2005-03-05 12:15
[prompt]$
```

Figura 7.9: Exemplos de uso do `at`.

REFERÊNCIAS BIBLIOGRÁFICAS

[Bailey (1998)] Bailey, E. C. *Maximum RPM*, Red Hat, 1998.

[Barnes (1999)] Barnes, D. *Rpm howto*, URL `http://www.rpm.org/RPM-HOWTO/`, 1999.

[Bergo (2001)] Bergo, F. P. G. *Autotut - using gnu auto{conf,make,header}*, URL `http://www.seul.org/docs/autotut`, 2001.

[Budlong (1999)] Budlong, M. *Dicas de boas práticas de programação shell*, URL `http://geocities.yahoo.com.br/cesarakg/tips-shell-programming.html`, 1999.

[Cooper (2005)] Cooper, M. *Advanced Bash-Scripting Guide*, URL `http://www.tldp.org/LDP/abs/`, 2005.

[Dias (2000)] Dias, R. S. *Revista do Linux*, Curitiba, Julho de 2000. p. 52-60.

[Dougherty & Robbins (1997)] Dougherty, D & Robbins, A. *Sed & Awk*, 2nd Edition, O'Reilly, 1997

[Friedl (2002)] Friedl, J. E. F. *Mastering Regular Expressions*, 2rd Edition, O'Reilly, 2002.

[FSF (2003)] Free Software Foundation. *The GNU Awk User's Guide*, 3rd Edition, URL `http://www.gnu.org/software/gawk/manual/gawk.html`, 2003.

[Jargas (2002)] Jargas, A. M. *Expressões Regulares*, URL `http://guia-er.sourceforge.net/`, 2002.

[Jargas (2003)] Jargas, A. M. *Sed-Howto*, URL `http://www.aurelio.net/sed/sed-HOWTO/`, 2002.

[Jargas (2004)] Jargas, A. M. *Shell Script - Por que programar tem que ser divertido*, URL `http://www.aurelio.net/shell/`, 2004.

[Jargas (2005)] Jargas, A. M. *Aurélio :: Sed*, URL `http://www.aurelio.net/sed/`, 2005.

[Michael (2003)] Michael, R. K. *Dominando Unix Shell Scripting*, Campus, 2003.

[Neves (2003)] Neves, J. C. *Programação Shell Linux*, Terceira Edição, Brasport, 2003.

[Pizzini (1998)] Pizzini, K. *sed, a stream editor*, URL `http://www.gnu.org/software/sed/manual/sed.html`, 1998

[Robbins (2001)] Robbins, A. *Effective Awk Programming*, 3rd Edition, O'Reilly, 2001. .

[Schneider (2003)] Schneider, B. O. *Desenvolvimento de Scripts e Pacotes*, Curso de Pós Graduação "Lato Sensu" (Especialização) a Distância em Administração em Redes Linux, UFLA/FAEPE, Lavras, 2003.

[Schwartz & Phoenix (2001)] Schwartz, R. L. & Phoenix, T. *Learning Perl*, 3rd Edition, O'Reilly, 2001.

[Vaughan, et al. (2001)] Vaughan, G. V.; Elliston, B.; Tromey, T. and Taylor, I. L. *GNU Autoconf, Automake, and Libtool*, New Riders, 2001.

[Wall, et al. (2000)] Wall, L.; Christiansem, T. and Orwant, J. *Programming Perl*, 3rd Edition, O'Reilly, 2000.