Shell Programming in Unix, Linux and OS X

Fourth Edition
Shell Programming
in Unix, Linux
and OS X

Fourth Edition

Stephen G. Kochan
Patrick Wood

Addison-Wesley
800 East 96th Street, Indianapolis, Indiana 46240
# Contents at a Glance

Introduction  1  

1  A Quick Review of the Basics  5  
2  What Is the Shell?  39  
3  Tools of the Trade  51  
4  And Away We Go  93  
5  Can I Quote You on That?  105  
6  Passing Arguments  121  
7  Decisions, Decisions  131  
8  'Round and 'Round She Goes  163  
9  Reading and Printing Data  185  
10  Your Environment  209  
11  More on Parameters  239  
12  Loose Ends  255  
13  Rolo Revisited  273  
14  Interactive and Nonstandard Shell Features  289  

A  Shell Summary  321  
B  For More Information  359  
Index  363
# Table of Contents

## Introduction 1

- How This Book Is Organized 2
- Accessing the Free Web Edition 3

## 1 A Quick Review of the Basics 5

### Some Basic Commands 5

- Displaying the Date and Time: The `date` Command 5
- Finding Out Who’s Logged In: The `who` Command 5
- Echoing Characters: The `echo` Command 6

### Working with Files 6

- Listing Files: The `ls` Command 7
- Displaying the Contents of a File: The `cat` Command 7
- Counting the Number of Words in a File: The `wc` Command 7
- Command Options 8
- Making a Copy of a File: The `cp` Command 8
- Renaming a File: The `mv` Command 8
- Removing a File: The `rm` Command 9

### Working with Directories 9

- The Home Directory and Pathnames 10
- Displaying Your Working Directory: The `pwd` Command 12
- Changing Directories: The `cd` Command 12
- More on the `ls` Command 15
- Creating a Directory: The `mkdir` Command 17
- Copying a File from One Directory to Another 18
- Moving Files Between Directories 19
- Linking Files: The `ln` Command 20
- Removing a Directory: The `rmdir` Command 23

### Filename Substitution 24

- The Asterisk 24
- Matching Single Characters 25

### Filename Nuances 27

- Spaces in Filenames 27
- Other Weird Characters 28

### Standard Input/Output, and I/O Redirection 28

- Standard Input and Standard Output 28
Contents

sed 70
   The -n Option 72
   Deleting Lines 73
tr 74
   The -s Option 76
   The -d Option 77
grep 78
   Regular Expressions and grep 81
   The -v Option 82
   The -l Option 82
   The -n Option 83
sort 84
   The -u Option 84
   The -r Option 85
   The -o Option 85
   The -n Option 86
   Skipping Fields 87
   The -t Option 87
   Other Options 88
uniq 88
   The -d Option 89
   Other Options 90

4 And Away We Go 93
   Command Files 93
   Comments 96
   Variables 97
      Displaying the Values of Variables 98
      Undefined Variables Have the Null Value 100
      Filename Substitution and Variables 101
      The ${variable} Construct 102
   Built-in Integer Arithmetic 103

5 Can I Quote You on That? 105
   The Single Quote 105
   The Double Quote 109
   The Backslash 111
Using the Backslash for Continuing Lines 112
The Backslash Inside Double Quotes 112
Command Substitution 114
The Back Quote 114
The {...} Construct 115
The expr Command 119

6 Passing Arguments 121
The $# Variable 122
The $* Variable 123
A Program to Look Up Someone in the Phone Book 124
A Program to Add Someone to the Phone Book 125
A Program to Remove Someone from the Phone Book 127
${n} 128
The shift Command 128

7 Decisions, Decisions 131
Exit Status 131
The $? Variable 132
The test Command 135
String Operators 135
An Alternative Format for test 139
Integer Operators 140
File Operators 142
The Logical Negation Operator ! 143
The Logical AND Operator -a 143
Parentheses 144
The Logical OR Operator -o 144
The else Construct 145
The exit Command 147
A Second Look at the rem Program 147
The elif Construct 148
Yet Another Version of rem 151
The case Command 153
Special Pattern-Matching Characters 155
The -x Option for Debugging Programs 157
Back to the case 159
Contents

The Null Command : 160
The && and || Constructs 161

8 'Round and 'Round She Goes 163
The for Command 163
   The $$ Variable 166
   The for Without the List 167
The while Command 168
The until Command 170
More on Loops 174
   Breaking Out of a Loop 174
   Skipping the Remaining Commands in a Loop 176
   Executing a Loop in the Background 177
   I/O Redirection on a Loop 177
   Piping Data into and out of a Loop 178
   Typing a Loop on One Line 179
The getopts Command 180

9 Reading and Printing Data 185
The read Command 185
   A Program to Copy Files 185
   Special echo Escape Characters 187
   An Improved Version of mycp 188
   A Final Version of mycp 190
   A Menu-Driven Phone Program 193
   The $$ Variable and Temporary Files 198
   The Exit Status from read 199
The printf Command 202

10 Your Environment 209
Local Variables 209
   Subshells 210
Exported Variables 211
   export -p 215
PS1 and PS2 216
HOME 217
PATH 217
Your Current Directory 225  
**CDPATH** 226  
More on Subshells 227  
*The . Command* 227  
*The exec Command* 230  
*The (...) and { ...; } Constructs* 231  
*Another Way to Pass Variables to a Subshell* 234  
*Your .profile File* 235  
*The TERM Variable* 236  
*The TZ Variable* 237  

11 More on Parameters 239  
*Parameter Substitution* 239  
`${parameter}` 239  
`${parameter}:-value` 240  
`${parameter}=value` 241  
`${parameter}?:value` 241  
`${parameter}:+value` 242  
*Pattern Matching Constructs* 242  
`${#variable}` 244  
*The $0 Variable* 245  
*The set Command* 246  
*The -x Option* 246  
*set with No Arguments* 247  
*Using set to Reassign Positional Parameters* 247  
*The -- Option* 248  
*Other Options to set* 251  
*The IFS Variable* 251  
*The readonly Command* 254  
*The unset Command* 254  

12 Loose Ends 255  
*The eval Command* 255  
*The wait Command* 257  
*The $! Variable* 257  
*The trap Command* 258  
*trap with No Arguments* 259
## Contents

Integer Arithmetic 303  
  Integer Types 304  
  Numbers in Different Bases 305  

The alias Command 307  
  Removing Aliases 309  

Arrays 309  

Job Control 315  
  Stopped Jobs and the fg and bg Commands 316  

Miscellaneous Features 317  
  Other Features of the cd Command 317  
  Tilde Substitution 318  
  Order of Search 319  

Compatibility Summary 319  

A Shell Summary 321  

Startup 321  

Commands 321  

Comments 322  

Parameters and Variables 322  
  Shell Variables 322  
  Positional Parameters 322  
  Special Parameters 323  
  Parameter Substitution 324  

Command Re-entry 326  
  The fc Command 326  
  vi Line Edit Mode 326  

Quoting 329  
  Tilde Substitution 329  
  Arithmetic Expressions 330  

Filename Substitution 331  

I/O Redirection 331  

Exported Variables and Subshell Execution 332  
  The (...) Construct 332  
  The { ...; } Construct 332  
  More on Shell Variables 333  

Functions 333  

Job Control 333
Shell Jobs 333
Stopping Jobs 334
Command Summary 334
The : Command 334
The . Command 334
The alias Command 335
The bg Command 335
The break Command 336
The case Command 336
The cd Command 337
The continue Command 338
The echo Command 338
The eval Command 339
The exec Command 339
The exit Command 340
The export Command 340
The false Command 341
The fc Command 341
The fg Command 342
The for Command 342
The getopt Command 343
The hash Command 344
The if Command 344
The jobs Command 347
The kill Command 347
The newgrp Command 347
The pwd Command 348
The read Command 348
The readonly Command 349
The return Command 349
The set Command 350
The shift Command 352
The test Command 352
The times Command 354
The trap Command 355
The true Command 356
The type Command 356
About the Authors

Stephen Kochan is the author or co-author of several best-selling titles on Unix and the C language, including Programming in C, Programming in Objective-C, Topics in C Programming, and Exploring the Unix System. He is a former software consultant for AT&T Bell Laboratories, where he developed and taught classes on Unix and C programming.

Patrick Wood is the CTO of the New Jersey location of Electronics for Imaging. He was a member of the technical staff at Bell Laboratories when he met Mr. Kochan in 1985. Together they founded Pipeline Associates, Inc., a Unix consulting firm, where he was vice president. They co-authored Exploring the Unix System, Unix System Security, Topics in C Programming, and Unix Shell Programming.
We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or email address.

Email: feedback@developers-library.info

Mail: Reader Feedback
Addison-Wesley Developer's Library
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.
This page intentionally left blank
It’s no secret that the family of Unix and Unix-like operating systems has emerged over the last few decades as the most pervasive, most widely used group of operating systems in computing today. For programmers who have been using Unix for many years, this came as no surprise: The Unix system provides an elegant and efficient environment for program development. That’s exactly what Dennis Ritchie and Ken Thompson sought to create when they developed Unix at Bell Laboratories way back in the late 1960s.

One of the strongest features of the Unix system is its wide collection of programs. More than 200 basic commands are distributed with the standard operating system and Linux adds to it, often shipping with 700–1000 standard commands! These commands (also known as tools) do everything from counting the number of lines in a file, to sending electronic mail, to displaying a calendar for any desired year.

But the real strength of the Unix system comes not from its large collection of commands but from the elegance and ease with which these commands can be combined to perform far more sophisticated tasks.

The standard user interface to Unix is the command line, which actually turns out to be a shell, a program that acts as a buffer between the user and the lowest levels of the system itself (the kernel). The shell is simply a program that reads in the commands you type and converts them into a form more readily understood by the system. It also includes core programming constructs that let you make decisions, loop, and store values in variables.

The standard shell distributed with Unix systems derives from AT&T’s distribution, which evolved from a version originally written by Stephen Bourne at Bell Labs. Since then, the IEEE has created standards based on the Bourne shell and the other more recent shells. The current version of this standard, as of this writing, is the Shell and Utilities volume of IEEE Std 1003.1-2001, also known as the POSIX standard. This shell is what we use as the basis for the rest of this book.

The examples in this book were tested on a Mac running Mac OS X 10.11, Ubuntu Linux 14.0, and an old version of SunOS 5.7 running on a Sparcstation Ultra-30. All examples, with the
exception of some Bash examples in Chapter 14, were run using the Korn shell, although all of them also work fine with Bash.

Because the shell offers an interpreted programming language, programs can be written, modified, and debugged quickly and easily. We turn to the shell as our first choice of programming language and after you become adept at shell programming, you will too.

**How This Book Is Organized**

This book assumes that you are familiar with the fundamentals of the system and command line; that is, that you know how to log in; how to create files, edit them, and remove them; and how to work with directories. In case you haven't used the Linux or Unix system for a while, we'll examine the basics in Chapter 1, “A Quick Review of the Basics.” In addition, filename substitution, I/O redirection, and pipes are also reviewed in the first chapter.

Chapter 2, “What Is the Shell?,” reveals what the shell really is, how it works, and how it ends up being your primary method of interacting with the operating system itself. You'll learn about what happens every time you log in to the system, how the shell program gets started, how it parses the command line, and how it executes other programs for you. A key point made in Chapter 2 is that the shell is just another program; nothing more, nothing less.

Chapter 3, “Tools of the Trade,” provides tutorials on tools useful in writing shell programs. Covered in this chapter are `cut`, `paste`, `sed`, `grep`, `sort`, `tr`, and `uniq`. Admittedly, the selection is subjective, but it does set the stage for programs that we'll develop throughout the remainder of the book. Also in Chapter 3 is a detailed discussion of regular expressions, which are used by many Unix commands, such as `sed`, `grep`, and `ed`.

Chapters 4 through 9 teach you how to put the shell to work for writing programs. You'll learn how to write your own commands; use variables; write programs that accept arguments; make decisions; use the shell's `for`, `while`, and `until` looping commands; and use the `read` command to read data from the terminal or from a file. Chapter 5, “Can I Quote you on That?”, is devoted entirely to a discussion of one of the most intriguing (and often confusing) aspects of the shell: the way it interprets quotes.

By that point in the book, all the basic programming constructs in the shell will have been covered, and you will be able to write shell programs to solve your particular problems.

Chapter 10, “Your Environment,” covers a topic of great importance for a real understanding of the way the shell operates: the environment. You'll learn about local and exported variables; subshells; special shell variables, such as `HOME`, `PATH`, and `CDPATH`; and how to set up your `.profile` file.

Chapter 11, “More on Parameters,” and Chapter 12, “Loose Ends,” tie up some loose ends, and Chapter 13, “Rolo Revisited,” presents a final version of a phone directory program called `rolo` that is developed throughout the book.
Chapter 14, “Interactive and Nonstandard Shell Features,” discusses features of the shell that either are not formally part of the IEEE POSIX standard shell (but are available in most Unix and Linux shells) or are mainly used interactively instead of in programs.

Appendix A, “Shell Summary,” summarizes the features of the IEEE POSIX standard shell.

Appendix B, “For More Information,” lists references and resources, including the Web sites where different shells can be downloaded.

The philosophy this book uses is to teach by example. We believe that properly chosen examples do a far better job of illustrating how a particular feature is used than ten times as many words. The old “A picture is worth …” adage seems to apply just as well to coding.

We encourage you to type in each example and test it on your own system, for only by doing can you become adept at shell programming. Don’t be afraid to experiment. Try changing commands in the program examples to see the effect, or add different options or features to make the programs more useful or robust.

Accessing the Free Web Edition

Your purchase of this book in any format includes access to the corresponding Web Edition, which provides several special features to help you learn:

- The complete text of the book online
- Interactive quizzes and exercises to test your understanding of the material
- Updates and corrections as they become available

The Web Edition can be viewed on all types of computers and mobile devices with any modern web browser that supports HTML5.

To get access to the Web Edition of Shell Programming with Unix, Linux, and OS X all you need to do is register this book:

2. Sign in or create a new account.
4. Answer the questions as proof of purchase.

The Web Edition will appear under the Digital Purchases tab on your Account page. Click the Launch link to access the product.
This page intentionally left blank
Tools of the Trade

This chapter provides detailed descriptions of some commonly used shell programming tools. Covered are `cut`, `paste`, `sed`, `tr`, `grep`, `uniq`, and `sort`. The more proficient you become at using these tools, the easier it will be to write efficient shell scripts.

Regular Expressions

Before getting into the tools, you need to learn about regular expressions. Regular expressions are used by many different Unix commands, including `ed`, `sed`, `awk`, `grep`, and, to a more limited extent, the `vi` editor. They provide a convenient and consistent way of specifying patterns to be matched.

Where this gets confusing is that the shell recognizes a limited form of regular expressions with filename substitution. Recall that the asterisk (*) specifies zero or more characters to match, the question mark (?) specifies any single character, and the construct `[...]` specifies any character enclosed between the brackets. But that's not the same thing as the more formal regular expressions we'll explore. For example, the shell sees `?` as a match for any single character, while a regular expression—commonly abbreviated regex—uses a period (.) for the same purpose.

True regular expressions are far more sophisticated than those recognized by the shell and there are entire books written about how to assemble really complex regex statements. Don't worry, though, you won't need to become an expert to find great value in regular expressions!

Throughout this section, we assume familiarity with a line-based editor such as `ex` or `ed`. See Appendix B for more information on these editors if you're not familiar with them, or check the appropriate `man` page.

Matching Any Character: The Period (.)

A period in a regular expression matches any single character, no matter what it is. So the regular expression

```
.$
```

matches an `r` followed by any single character.
The regular expression
\.x\.
matches an x that is surrounded by any two characters, not necessarily the same.

We can demonstrate a lot of regular expressions by using the simple ed editor, an old-school line-oriented editor that has been around as long as Linux have been around.

For example, the ed command

```
/ \.../ 
```

searches forward in the file you are editing for the first line that contains any three characters surrounded by blanks. But before we demonstrate that, notice in the very beginning of this example that ed shows the number of characters in the file (248) and that commands like print (p) can be prefixed with a range specifier, with the most basic being 1, $, which is the first through last line of the file:

```
$ ed intro
248
1,$p
```

Print all the lines

The Unix operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in the design of the Unix system was to create an environment that promoted efficient program development.

That's our working file. Now let's try some regular expressions:

```
/ \.../ 
```

Look for three chars surrounded by blanks

The Unix operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in the design of the Unix system was to create an environment that promoted efficient program development.

In the first search, ed started searching from the beginning of the file and found that the sequence “was” in the first line matched the indicated pattern and printed it.

Repeating the search (the ed command /) resulted in the display of the second line of the file because “and” matched the pattern. The substitute command s that followed specified that all occurrences of the character p, followed by any single character, followed by the character o were to be replaced by the characters XXX. The prefix 1,$ indicates that it should be applied to all lines in the file, and the substitution is specified with the structure s/old/new/g, where s
indicates it’s a substitution, the slashes delimit the old and new values, and \texttt{g} indicates it should be applied as many times as needed for each line, not just once per line.

**Matching the Beginning of the Line: The Caret (\texttt{^})**

When the caret character \texttt{^} is used as the first character in a regular expression, it matches the beginning of the line. So the regular expression

\texttt{^George}

matches the characters \texttt{George only if they occur at the beginning of the line}. This is actually known as “left-rooting” in the regex world, for obvious reasons.

Let’s have a look:

```bash
$ ed intro
248 /the/
>>in the late 1960s. One of the primary goals in
>>the design of the Unix system was to create an
/^the/ The design of the Unix system was to create an
1,$s/^/>>/ Insert >> at the beginning of each line
1,$p
>>The Unix operating system was pioneered by Ken
>>Thompson and Dennis Ritchie at Bell Laboratories
>>in the late 1960s. One of the primary goals in
>>the design of the Unix system was to create an
>>environment that promoted efficient program
>>development.
```

The preceding example also shows how the regular expression \texttt{^} can be used to match the beginning of the line. Here it is used to insert the characters \texttt{>>} at the start of each line. A command like

```bash
1,$s/^/    / Insert spaces at the start of each line
```

is also commonly used to insert spaces at the start of each line (in this case four spaces would be inserted).

**Matching the End of the Line: The Dollar Sign ($)**

Just as the \texttt{^} is used to match the beginning of the line, so the dollar sign \texttt{$} is used to match the end of the line. So the regular expression

```bash
contents$
```

matches the characters \texttt{contents only if they are the last characters on the line}. What do you think would be matched by the regular expression

\texttt{.}$
Would this match a period character that ends a line? No. Recall that the period matches any character, so this would match any single character at the end of the line (including a period).

So how do you match a period? In general, if you want to match any of the characters that have a special meaning in regular expressions, precede the character by a backslash (\) to override its special meaning. For example, the regular expression

```
\.$
```

matches any line that ends in a period, and the regular expression

```
^\.
```

matches any line that starts with a period.

Want to specify a backslash as an actual character? Use two backslashes in a row: `\\`.

```
$ ed intro
248
/
\.$/                  \  \  \  \  \  \  \  Search for a line that ends with a period
development.
1,\.$/$$/\        Add >> to the end of each line
1,$p
```

The Unix operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in the design of the Unix system was to create an environment that promoted efficient program development.

```
1,\.$/..$/ //        Delete the last two characters from each line
1,$p
```

The Unix operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in the design of the Unix system was to create an environment that promoted efficient program development.

A common use of `^` and `$` is the regular expression

```
^\$
```

which matches any line that contains no characters at all. Note that this regular expression is different from

```
^ $\n```

which matches any line that consists of a single space character.
Matching a Character Set: The \[. . . \] Construct

Suppose that you are editing a file and want to search for the first occurrence of the characters the. In ed, this is easy: You simply type the command

```
/the/
```

This causes ed to search forward in its buffer until it finds a line containing the indicated sequence. The first line that matches will be displayed by ed:

```
$ ed intro
248 /the/ Find line containing the
in the late 1960s. One of the primary goals in
```

Notice that the first line of the file also contains the word the, except it begins with a capital T. A regular expression that searches for either the or The can be built using a character set: the characters \[ and \] can be used to specify that one of the enclosed character set is to be matched. The regular expression

```
[tT]he
```

would match a lower- or uppercase t followed immediately by the characters he:

```
$ ed intro
248 /[tT]he/ Look for the or The
The Unix operating system was pioneered by Ken
/ Continue the search
in the late 1960s. One of the primary goals in
/ Once again
the design of the Unix system was to create an
1,$s/[aeiouAEIOU]//g Delete all vowels
1,$p
Th nx prtng systm ws pnrd by Kn
Thmpsn nd Dnns Rcht t Bll Lbrtrs
n th lt 1960s. n f th prmy gls n
th dgn f th nx systm ws t crt n
nvrmnt th prmd ffcnt prgrm
dvlpmnt.
```

Notice the example in the above of \[aeiouAEIOU\] which will match a single vowel, either uppercase or lowercase. That notation can get rather clunky, however, so a range of characters can be specified inside the brackets instead. This can be done by separating the starting and ending characters of the range by a dash (-). So, to match any digit character 0 through 9, you could use the regular expression

```
[0-9]
```

or, more succinctly, you could write

```
[0-9]
```
To match an uppercase letter, use [A-Z]

To match an upper- or lowercase letter, you write [A-Za-z]

Here are some examples with ed:

$ ed intro
248
/[0-9]/          Find a line containing a digit
in the late 1960s. One of the primary goals in
/^[A-Z]/           Find a line that starts with an uppercase letter
The Unix operating system was pioneered by Ken
/
Again
Thompson and Dennis Ritchie at Bell Laboratories
1,$s/[A-Z]/*/g    Change all uppercase letters to *s
1,$p
*he *nix operating system was pioneered by *en
*thompson and *ennis *itchie at *ell *aboratories
in the late 1960s. *ne of the primary goals in
the design of the *nix system was to create an
environment that promoted efficient program
development.

As you’ll learn below, the asterisk is a special character in regular expressions. However, you
don’t need to put a backslash before it in the replacement string of the substitute command
because the substitution’s replacement string has a different expression language (we did
mention that this can be a bit tricky at times, right?).

In the ed editor, regular expression sequences such as *, .., [...], $, and ^ are only meaningful
in the search string and have no special meaning when they appear in the replacement string.

If a caret (^) appears as the first character after the left bracket, the sense of the match is
inverted. (By comparison, the shell uses the ! for this purpose with character sets.) For example,
the regular expression
[^A-Z]
matches any character except an uppercase letter. Similarly,
[^A-Za-z]
matches any non-alphabetic character. To demonstrate, let’s remove all non-alphabetic charac-
ters from the lines in our test file:

$ ed intro
248
1,$s/[a-zA-Z]//g    Delete all non-alphabetic characters
1,$p
TheUnixoperatingsystemwaspioneeredbyKen
ThompsonandDennisRitchieatBellLaboratories
Matching Zero or More Characters: The Asterisk (*)

The asterisk is used by the shell in filename substitution to match zero or more characters. In forming regular expressions, the asterisk is used to match zero or more occurrences of the preceding element of the regular expression (which may itself be another regular expression).

So, for example, the regular expression

\( X^* \)

matches zero, one, two, three, ... capital X's while the expression

\( XX^* \)

matches one or more capital X's, because the expression specifies a single X followed by zero or more X's. You can accomplish the same effect with a + instead: it matches one or more of the preceding expression, so \( XX^* \) and \( X^+ \) are identical in function.

A similar type of pattern is frequently used to match one or more blank spaces in a line:

```
$ ed lotsaspaces
85
1.$p
This is an example of a file that contains a lot of blank spaces Change multiple blanks to single blanks
1,$s/ */ /g
1.$p
This is an example of a file that contains a lot of blank spaces
```

The `ed` command

```
1,$s/ */ /g
```

told the program to substitute all occurrences of a space followed by zero or more spaces with a single space—in other words, to collapse all whitespace into single spaces. If it matches a single space, there’s no change. But if it matches three spaces, say, they’ll all be replaced by a single space.

The regular expression

\( .^* \)

is often used to specify zero or more occurrences of *any* characters. Bear in mind that a regular expression matches the longest string of characters that match the pattern. Therefore, used by itself, this regular expression always matches the *entire* line of text.
As another example of the combination of . and *, the regular expression
\[e.*e\]
matches all the characters from the first e on a line to the last one.

*Note that it doesn’t necessarily match only lines that start and end with an e, however, because it’s not left- or right-rooted (that is, it doesn’t use ^ or $ in the pattern).*

```
$ ed intro
248
1.$s/e.*e/+++/
1.$p
Th+++n
Thompson and D+++S
in th+++ primary goals in
th+++ an
+++nt program
d+++nt.
```

Here’s an interesting regular expression. What do you think it matches?

\[[A-Za-z][A-Za-z]*\]

This matches any alphabetic character followed by zero or more alphabetic characters. This is pretty close to a regular expression that matches words and can be used as shown below to replace all words with the letter X while retaining all spaces and punctuation.

```
$ ed intro
248
1.$s/[A-Za-z][A-Za-z]*/X/g
1.$p
X X X X X X X X
X X X X X X X
X X X X X X X X
X X X X X
X X.
```

The only thing it didn’t match in this example was the numeric sequence 1960. You can change the regular expression to also consider a sequence of digits as a word too, of course:

```
$ ed intro
248
1.$s/[A-Za-z0-9][A-Za-z0-9]*/X/g
1.$p
X X X X X X X X
X X X X X X
X X X X X X X X
X X X X X X X X
X X X X
X.
```
We could expand on this to consider hyphenated and contracted words (for example, *don’t*), but we’ll leave that as an exercise for you. As a point to note, if you want to match a dash character inside a bracketed choice of characters, you must put the dash immediately after the left bracket (but after the inversion character `^` if present) or immediately before the right bracket for it to be properly understood. That is, either of these expressions

```
[-0-9]
[0-9-]
```

matches a single dash or digit character.

In a similar fashion, if you want to match a right bracket character, it must appear after the opening left bracket (and after the `^` if present). So

```
[] a-z]
```

matches a right bracket or a lowercase letter.

**Matching a Precise Number of Subpatterns: `\{ . . \}`**

In the preceding examples, you saw how to use the asterisk to specify that *one or more* occurrences of the preceding regular expression are to be matched. For instance, the regular expression

```
XX*
```

means match an *X* followed by zero or more subsequent occurrences of the letter *X*. Similarly,

```
XXX*
```

means match at least *two* consecutive *X*'s.

Once you get to this point, however, it ends up rather clunky, so there is a more general way to specify a precise number of characters to be matched: by using the construct

```
\{min, max\}
```

where *min* specifies the minimum number of occurrences of the preceding regular expression to be matched, and *max* specifies the maximum. Notice that you need to *escape* the curly brackets by preceding each with a backslash.

The regular expression

```
x\{1,10\}
```

matches from one to 10 consecutive *x*'s. Whenever there’s a choice, the largest pattern is matched, so if the input text contains eight consecutive *x*'s, that is how many will be matched by the preceding regular expression.

As another example, the regular expression

```
[A-Za-z]\{4,7\}
```

matches a sequence of alphabetic letters from four to seven characters long.
Let's try a substitution using this notation:

```
$ ed intro
248
1,$s/[A-Za-z]\{4,7\}/X/g
1,$p
```

The X Xng X was Xed by Ken
Xn and X X at X XX
in the X 1960s. One of the X X in
the X of the X X was to X an
XX X Xd Xnt X
XX.

This invocation is a specific instance of a global search and replace in `ed` (and, therefore, also in `vi`): `s/old/new/`. In this case, we add a range of `1,$` beforehand and the `g` flag is appended to ensure that multiple substitutions will occur on each line, as appropriate.

A few special cases of this special construct are worth noting. If only one number is enclosed by braces, as in
```
\{10\}
```
that number specifies that the preceding regular expression must be matched *exactly* that many times. So
```
[a-zA-Z]\{7\}
```
matches exactly seven alphabetic characters; and
```
.\{10\}
```
matches exactly 10 characters no matter what they are:

```
$ ed intro
248
1,$s/\.\{5\}$///
1,$p
```

Operating system was pioneered by Ken
nd Dennis Ritchie at Bell Laboratories
e 1960s. One of the primary goals in
of the Unix system was to create an
 that promoted efficient program
n.

```
1,$s/\.\{5\}$///
1,$p
```

Operating system was pioneered b
nd Dennis Ritchie at Bell Laborat
e 1960s. One of the primary goa
of the Unix system was to crea
n that promoted efficient pr
n.
Note that the last line of the file didn’t have five characters when the last substitute command was executed; therefore, the match failed on that line and thus was left alone because we specified that exactly five characters were to be deleted.

If a single number is enclosed in the braces, followed immediately by a comma, then at least that many occurrences of the previous regular expression must be matched, but no upper limit is set. So

\+\{5,\}

matches at least five consecutive plus signs. If more than five occur sequentially in the input data, the largest number is matched.

$\texttt{ed intro}$

248

1.$\texttt{s/[a-zA-Z]\{6,\}/X/g} \quad \text{Change words at least 6 letters long to X}$

1.\p

The Unix X X was X by Ken X and X X at Bell X in the late 1960s. One of the X goals in the X of the Unix X was to X an X that X X X X.

Saving Matched Characters: $\texttt{\( \ldots \)}$

It is possible to reference the characters matched against a regular expression by enclosing those characters inside backslashed parentheses. These captured characters are stored in pre-defined variables in the regular expression parser called registers, which are numbered 1 through 9.

This gets a bit confusing, so take this section slowly!

As a first example, the regular expression

^\(\.\)$

matches the first character on the line, whatever it is, and stores it into register 1.

To retrieve the characters stored in a particular register, the construct \texttt{\n} is used, where \texttt{n} is a digit from 1 to 9. So the regular expression

^\(\.\)\n$

initially matches the first character on the line and stores it in register 1, then matches whatever is stored in register 1, as specified by the \n. The net effect of this regular expression is to match the first two characters on a line if they are both the same character. Tricky, eh?

The regular expression

^\(\.\)\.*$\n
matches all lines in which the first character on the line (\texttt{^\.)} is the same as the last character on the line (\texttt{\$}). The \texttt{.*} matches all the characters in-between.
Let's break this one down. Remember ^ is the beginning of line and $ the end of line. The simplified pattern is then ..* which is the first character of the line (the first .) followed by the .* for the rest of the line. Add the \( \) notation to push that first character into register 1 and 1 to then reference the character, and it should make sense to you.

Successive occurrences of the \( \)\( ... \)\( \) construct get assigned to successive registers. So when the following regular expression is used to match some text

\( ^{\( \)\( ... \)\( \)\( \)\( ... \)\( \)} \)

the first three characters on the line will be stored into register 1, and the next three characters into register 2. If you appended \( 2\)1 to the pattern, you would match a 12-character string in which characters 1–3 matched characters 10–12, and in which characters 4–6 matched characters 7–9.

When using the substitute command in ed, a register can also be referenced as part of the replacement string, which is where this can be really powerful:

$ ed phonebook
114
1,$p
Alice Chebba 973-555-2015
Barbara Swingle 201-555-9257
Liz Stachiw 212-555-2298
Susan Goldberg 201-555-7776
Tony Iannino 973-555-1295
1,$s/\(.*\)\(.*\)/\2 \1/ Switch the two fields
1,$p
973-555-2015 Alice Chebba
201-555-9257 Barbara Swingle
212-555-2298 Liz Stachiw
201-555-7776 Susan Goldberg
973-555-1295 Tony Iannino

The names and the phone numbers are separated from each other in the phonebook file by a single tab character. The regular expression

\( \(.*\) \(.*\) \)

says to match all the characters up to the first tab (that's the character sequence .* between the \( and the \) ) and assign them to register 1, and to match all the characters that follow the tab character and assign them to register 2. The replacement string

\2 \1

specifies the contents of register 2, followed by a space, followed by the contents of register 1.

When ed applies the substitute command to the first line of the file:

Alice Chebba 973-555-2015
it matches everything up to the tab (Alice Chebba) and stores it into register 1, and everything after the tab (973-555-2015) and stores it into register 2. The tab itself is lost because it’s not surrounded by parentheses in the regex. Then ed substitutes the characters that were matched (the entire line) with the contents of register 2 (973-555-2015), followed by a space, followed by the contents of register 1 (Alice Chebba):

973-555-2015 Alice Chebba

As you can see, regular expressions are powerful tools that enable you to match and manipulate complex patterns, albeit with a slight tendency to look like a cat ran over your keyboard at times!

Table 3.1 summarizes the special characters recognized in regular expressions to help you understand any you encounter and so you can build your own as needed.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Example</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Any character</td>
<td>a..</td>
<td>a followed by any two characters</td>
</tr>
<tr>
<td>^</td>
<td>Beginning of line</td>
<td>^wood</td>
<td>wood only if it appears at the beginning of the line</td>
</tr>
<tr>
<td>$</td>
<td>End of line</td>
<td>x$</td>
<td>x only if it is the last character on the line</td>
</tr>
<tr>
<td>^INSERT$</td>
<td>A line containing just the characters INSERT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^$</td>
<td>A line that contains no characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Zero or more occurrences of previous regular expression</td>
<td>x*</td>
<td>Zero or more consecutive x’s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xx*</td>
<td>One or more consecutive x’s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.*</td>
<td>Zero or more characters w followed by zero or more characters followed by an s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w.*$</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>One or more occurrences of previous regular expression</td>
<td>x+</td>
<td>One or more consecutive x’s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xx+</td>
<td>Two or more consecutive x’s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.+</td>
<td>One or more characters w followed by one or more characters followed by an s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w.+$</td>
<td></td>
</tr>
<tr>
<td>[chars]</td>
<td>Any character in chars</td>
<td>[tT]</td>
<td>Lower- or uppercase t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[a-z]</td>
<td>Lowercase letter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[a-zA-Z]</td>
<td>Lowercase letter</td>
</tr>
<tr>
<td>[^chars]</td>
<td>Any character not in chars</td>
<td>[^0-9]</td>
<td>Any non-numeric character</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[^a-zA-Z]</td>
<td>Any non-alphabetic character</td>
</tr>
</tbody>
</table>

(Continued)
Chapter 3  Tools of the Trade

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Example</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>({\text{min}, \text{max}})</td>
<td>At least (\text{min}) and at most (\text{max}) occurrences of previous regular expression</td>
<td>(x\{1, 5})</td>
<td>At least 1 and at most 5 x’s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([0-9]\{3, 9})</td>
<td>Anywhere from 3 to 9 successive digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([0-9]{3})</td>
<td>Exactly 3 digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([0-9]{3,})</td>
<td>At least 3 digits</td>
</tr>
</tbody>
</table>

\(\{\ldots\}\)  
Save characters matched between parentheses in next register (1-9)  
^\(\ldots\)  
First character on the line; stores it in register 1  
^\(\ldots\)\1  
First and second characters on the line if they’re the same  
^\(\ldots\)\(\ldots\)  
First and second characters on the line; stores first character in register 1 and second character in register 2

\textbf{cut}

This section teaches you about a useful command known as \texttt{cut}. This command comes in handy when you need to extract (that is, “cut out”) various fields of data from a data file or the output of a command. The general format of the \texttt{cut} command is

\texttt{cut -cchars file}

where \texttt{chars} specifies which characters (by position) you want to extract from each line of \texttt{file}. This can consist of a single number, as in \texttt{-c5} to extract the fifth character from each line of input; a comma-separated list of numbers, as in \texttt{-c1,13,50} to extract characters 1, 13, and 50; or a dash-separated range of numbers, as in \texttt{-c20-50} to extract characters 20 through 50, inclusive. To extract characters to the end of the line, you can omit the second number of the range so

\texttt{cut -c5- data}

extracts characters 5 through the end of the line from each line of \texttt{data} and writes the results to standard output.

If \texttt{file} is not specified, \texttt{cut} reads its input from standard input, meaning that you can use \texttt{cut} as a filter in a pipeline.

Let’s take another look at the output from the \texttt{who} command:

\$ \texttt{who}
root  console Feb 24 08:54
steve  tty02 Feb 24 12:55
george  tty08 Feb 24 09:15
dawn  tty10 Feb 24 15:55
\$

\$
As shown, four people are logged in. Suppose that you just want to know the names of the logged-in users and don’t care about what terminals they are on or when they logged in. You can use the `cut` command to cut out just the usernames from the `who` command’s output:

```
$ who | cut -c1-8
root
steve
george
dawn
$
```

The `-c1-8` option to `cut` specifies that characters 1 through 8 are to be extracted from each line of input and written to standard output.

The following shows how you can tack a `sort` to the end of the preceding pipeline to get a sorted list of the logged-in users:

```
$ who | cut -c1-8 | sort
dawn
george
root
steve
$
```

Note, this is our first three-command pipe. Once you get the concept of output connected to subsequent input, pipes of three, four or more commands are logical and easy to assemble.

If you wanted to see which terminals were currently being used or which pseudo or virtual terminals were in use, you could cut out just the `tty` field from the `who` command output:

```
$ who | cut -c10-16
console
tty02
tty08
tty10
$
```

How did you know that `who` displays the terminal identification in character positions 10 through 16? Simple! You executed the `who` command at your terminal and counted out the appropriate character positions.

You can use `cut` to extract as many different characters from a line as you want. Here, `cut` is used to display just the username and login time of all logged-in users:

```
$ who | cut -c1-8,18-
root Feb 24 08:54
steve Feb 24 12:55
george Feb 24 09:15
dawn Feb 24 15:55
$
```
The option `-c1-8,18-` specifies "extract characters 1 through 8 (the username) and also characters 18 through the end of the line (the login time)."

The `-d` and `-f` Options

The `cut` command with its `-c` flag is useful when you need to extract data from a file or command, provided that file or command has a fixed format.

For example, you could use `cut` with the `who` command because you know that the usernames are always displayed in character positions 1–8, the terminal in 10–16, and the login time in 18–29. Unfortunately, not all your data will be so well organized!

For instance, take a look at the `/etc/passwd` file:

```
$ cat /etc/passwd
root:*:0:0:The Super User::usr/bin/ksh
cron:*:1:1:Cron Daemon for periodic tasks::
bin:*:3:3:The owner of system files::
ucp:*:5:5:/usr/spool/uucp:/usr/lib/uucp/uucico
asg:*:6:6:The Owner of Assignable Devices::
steve:*:203:100::/users/steve:/usr/bin/ksh
other:*:4:4:Needed by secure program::
```

`/etc/passwd` is the master file that contains the usernames of all users on your computer system. It also contains other information such as user ID, home directory, and the name of the program to start up when that particular user logs in.

Quite clearly, the data in this file does not line up anywhere near as neatly as the `who`'s output does. Therefore extracting a list of all the users of your system from this file cannot be done using the `-c` option to `cut`.

Upon closer inspection of the file, however, it's clear that fields are separated by a colon character. Although each field may not be the same length from one line to the next, you can "count colons" to get the same field from each line.

The `-d` and `-f` options are used with `cut` when you have data that is delimited by a particular character, with `-d` specifying the field separator delimiter and `-f` the field or fields you want extracted. The invocation of the `cut` command becomes

```
cut -d: -f1 /etc/passwd
```

where `dchar` is the character that delimits each field of the data, and `fields` specifies the fields to be extracted from `file`. Field numbers start at 1, and the same type of formats can be used to specify field numbers as was used to specify character positions before (for example, `-f1,2,8,-f1-3,-f4-`).

To extract the names of all users from `/etc/passwd`, you could type the following:

```
$ cut -d: -f1 /etc/passwd
```

```
root
cron
bin
```
Given that the home directory of each user is in field 6, you can match up each user of the system with their home directory:

```bash
$ cut -d: -f1,6 /etc/passwd
root: /
cron: /
bin: /
uucp:/usr/spool/uucp
asg: /
steve:/users/steve
other: /
$
```

If the `cut` command is used to extract fields from a file and the `-d` option is not supplied, `cut` uses the tab character as the default field delimiter.

The following depicts a common pitfall when using the `cut` command. Suppose that you have a file called `phonebook` that has the following contents:

```bash
$ cat phonebook
Alice Chebba    973-555-2015
Barbara Swingle 201-555-9257
Jeff Goldberg   201-555-3378
Liz Stachiw     212-555-2298
Susan Goldberg  201-555-7776
Tony Iannino    973-555-1295
$
```

If you just want to get the names of the people in your phone book, your first impulse would be to use `cut` as shown:

```bash
$ cut -c1-15 phonebook
Alice Chebba    97
Barbara Swingle 2
Jeff Goldberg   2
Liz Stachiw     212
Susan Goldberg  2
Tony Iannino    97
$
```

Not quite what you want! This happened because the name is separated from the phone number by a tab character, not a set of spaces. As far as `cut` is concerned, tabs count as a single character when using the `-c` option. Therefore `cut` extracts the first 15 characters from each line, producing the results shown.
In a situation where the fields are separated by tabs, you should use the `-f` option to `cut` instead:

```
$ cut -f1 phonebook
Alice Chebba
Barbara Swingle
Jeff Goldberg
Liz Stachiw
Susan Goldberg
Tony Iannino
```

Recall that you don't have to specify the delimiter character with the `-d` option because `cut` defaults to a tab character delimiter.

How do you know in advance whether fields are delimited by blanks or tabs? One way to find out is by trial and error, as shown previously. Another way is to type the command `sed -n l file` at your terminal. If a tab character separates the fields, `\t` will be displayed instead of the tab:

```
$ sed -n l phonebook
Alice Chebba\t973-555-2015
Barbara Swingle\t201-555-9257
Jeff Goldberg\t201-555-3378
Liz Stachiw\t212-555-2298
Susan Goldberg\t201-555-7776
Tony Iannino\t973-555-1295
```

The output verifies that each name is separated from each phone number by a tab character. The stream editor `sed` is covered in more detail a bit later in this chapter.

**paste**

The `paste` command is the inverse of `cut`: Instead of breaking lines apart, it puts them together. The general format of the `paste` command is `paste files`

where corresponding lines from each of the specified `files` are “pasted” or merged together to form single lines that are then written to standard output. The dash character `-` can also be used in the `files` sequence to specify that input is from standard input.

Suppose that you have a list of names in a file called `names`:

```
$ cat names
Tony
Emanuel
Lucy
```
Ralph
Fred
$

Suppose that you also have a second file called **numbers** that contains corresponding phone numbers for each name in **names**:

```bash
$ cat numbers
(307) 555-5356
(212) 555-3456
(212) 555-9959
(212) 555-7741
(212) 555-0040
$
```

You can use `paste` to print the names and numbers side-by-side as shown:

```bash
$ paste names numbers
Tony    (307) 555-5356
Emanuel (212) 555-3456
Lucy    (212) 555-9959
Ralph   (212) 555-7741
Fred    (212) 555-0040
$
```

Each line from **names** is displayed with the corresponding line from **numbers**, separated by a tab.

The next example illustrates what happens when more than two files are specified:

```bash
$ cat addresses
55-23 Vine Street, Miami
39 University Place, New York
17 E. 25th Street, New York
38 Chauncey St., Bensonhurst
17 E. 25th Street, New York
$
```

```bash
$ paste names addresses numbers
Tony    55-23 Vine Street, Miami       (307) 555-5356
Emanuel 39 University Place, New York  (212) 555-3456
Lucy    17 E. 25th Street, New York    (212) 555-9959
Ralph   38 Chauncey St., Bensonhurst   (212) 555-7741
Fred    17 E. 25th Street, New York    (212) 555-0040
$
```

**The `-d` Option**

If you don't want the output fields separated by tab characters, you can specify the `-d` option to specify the output delimiter:

```bash
-d chars
```
where \textit{chars} is one or more characters that will be used to separate the lines pasted together. That is, the first character listed in \textit{chars} will be used to separate lines from the first file that are pasted with lines from the second file; the second character listed in \textit{chars} will be used to separate lines from the second file from lines from the third, and so on.

If there are more files than there are characters listed in \textit{chars}, paste “wraps around” the list of characters and starts again at the beginning.

In the simplest form of the -d option, specifying just a single delimiter character causes that character to be used to separate all pasted fields:

\begin{verbatim}
$ paste -d'+' names addresses numbers
Tony+55-23 Vine Street, Miami+(307) 555-5356
Emanuel+39 University Place, New York+(212) 555-3456
Lucy+17 E. 25th Street, New York+(212) 555-9959
Ralph+38 Chauncey St., Bensonhurst+(212) 555-7741
Fred+17 E. 25th Street, New York+(212) 555-0040
\end{verbatim}

Notice that it’s always safest to enclose the delimiter characters in single quotes. The reason why will be explained shortly.

The -s Option

The -s option tells paste to paste together lines from the same file, not from alternate files. If just one file is specified, the effect is to merge all the lines from the file together, separated by tabs, or by the delimiter characters specified with the -d option.

\begin{verbatim}
$ paste -s names
Tony Emanuel Lucy Ralph Fred
$ ls | paste -d' ' -s -
Paste ls’s output, use space as delimiter
addresses intro lotsaspace names numbers phonebook
$ 
\end{verbatim}

In the former example, the output from \textit{ls} is piped to \textit{paste} which merges the lines (-s option) from standard input (-), separating each field with a space (-d' ' option). You’ll recall from Chapter 1 that the command

\begin{verbatim}
echo *
\end{verbatim}

would have also listed all the files in the current directory, perhaps slightly less complicated than \textit{ls | paste}.

\textbf{sed}

\textit{sed} is a program used for editing data in a pipe or command sequence. It stands for \textit{stream editor}. Unlike \textit{ed}, \textit{sed} cannot be used interactively, though its commands are similar. The general form of the \textit{sed} command is

\begin{verbatim}
\textit{sed} command file
\end{verbatim}
where \texttt{command} is an \texttt{ed}-style command applied to each line of the specified \texttt{file}. If no file is specified, standard input is assumed.

As \texttt{sed} applies the indicated command or commands to each line of the input, it writes the results to standard output.

Let’s have a look. First, the \texttt{intro} file again:

\begin{verbatim}
$ cat intro
The Unix operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in the design of the Unix system was to create an environment that promoted efficient program development.
$
\end{verbatim}

Suppose that you want to change all occurrences of “Unix” in the text to “UNIX.” This can be easily done in \texttt{sed} as follows:

\begin{verbatim}
$ sed 's/Unix/UNIX/' intro       Substitute Unix with UNIX
The UNIX operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in the design of the UNIX system was to create an environment that promoted efficient program development.
$
\end{verbatim}

Get into the habit of enclosing your \texttt{sed} command in single quotes. Later, you’ll know when the quotes are necessary and when it’s better to use double quotes instead.

The \texttt{sed} command \texttt{s/Unix/UNIX/} is applied to every line of \texttt{intro}. Whether or not the line is modified, it gets written to standard output. Since it’s in the data stream also note that \texttt{sed} makes no changes to the original input file.

To make the changes permanent, you must redirect the output from \texttt{sed} into a temporary file and then replace the original file with the newly created one:

\begin{verbatim}
$ sed 's/Unix/UNIX/' intro > temp       Make the changes
$ mv temp intro                          And now make them permanent
$
\end{verbatim}

Always make sure that the correct changes were made to the file before you overwrite the original; a \texttt{cat} of \texttt{temp} would have been smart before the \texttt{mv} command overwrote the original data file.

If your text included more than one occurrence of “Unix” on a line, the above \texttt{sed} would have changed just the first occurrence to “UNIX.” By appending the \texttt{global} option \texttt{g} to the end of the substitute command \texttt{s}, you can ensure that multiple occurrences on a line will be changed.
In this case, the `sed` command would read

```
$ sed 's/Unix/UNIX/g' intro > temp
```

Now suppose that you wanted to extract just the usernames from the output of `who`. You already know how to do that with the `cut` command:

```
$ who | cut -c1-8
root
ruth
steve
pat
```

Alternatively, you can use `sed` to delete all the characters from the first space (which marks the end of the username) through the end of the line by using a regular expression:

```
$ who | sed 's/ .*$//'
root
ruth
steve
pat
```

The `sed` command substitutes a blank space followed by any characters up through the end of the line (`.*$`) with `nothing` (`//`); that is, it deletes the characters from the first blank to the end of the line for each input line.

**The `-n` Option**

By default, `sed` writes each line of input to standard output, whether or not it gets changed. Sometimes, however, you’ll want to use `sed` just to extract specific lines from a file. That’s what the `-n` flag is for: it tells `sed` that you don’t want it to print any lines by default. Paired with that, use the `p` command to print whichever lines match your specified range or pattern. For example, to print just the first two lines from a file:

```
$ sed -n '1,2p' intro
```

Just print the first 2 lines

The UNIX operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories

```
$ if, instead of line numbers, you precede the `p` command with a sequence of characters enclosed in slashes, `sed` prints just the lines from standard input that match that pattern. The following example shows how `sed` can be used to display just the lines that contain a particular string:

```
$ sed -n '/UNIX/p' intro
```

Just print lines containing UNIX

The UNIX operating system was pioneered by Ken

the design of the UNIX system was to create an

```
$
Deleting Lines

To delete lines of text, use the `d` command. By specifying a line number or range of numbers, you can delete specific lines from the input. In the following example, `sed` is used to delete the first two lines of text from `intro`:

```
$ sed '1,2d' intro
```

Delete lines 1 and 2

in the late 1960s. One of the primary goals in the design of the UNIX system was to create an environment that promoted efficient program development.

Remembering that by default `sed` writes all lines of the input to standard output, the remaining lines in text—that is, lines 3 through the end—simply get written to standard output.

By preceding the `d` command with a pattern, you can use `sed` to delete all lines that contain that text. In the following example, `sed` is used to delete all lines of text containing the word `UNIX`:

```
$ sed '/UNIX/d' intro
```

Delete all lines containing UNIX

Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in environment that promoted efficient program development.

The power and flexibility of `sed` goes far beyond what we’ve shown here. `sed` has facilities that enable you to loop, build text in a buffer, and combine many commands into a single editing script. Table 3.2 shows some more examples of `sed` commands.

<table>
<thead>
<tr>
<th><code>sed</code> Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sed '5d'</code></td>
<td>Delete line 5</td>
</tr>
<tr>
<td><code>sed '/[Tt]est/d'</code></td>
<td>Delete all lines containing <code>Test</code> or <code>test</code></td>
</tr>
<tr>
<td><code>sed -n '20,25p' text</code></td>
<td>Print only lines 20 through 25 from <code>text</code></td>
</tr>
<tr>
<td><code>sed '1,10s/unix/UNIX/g' intro</code></td>
<td>Change <code>unix</code> to <code>UNIX</code> wherever it appears in the first 10 lines of <code>intro</code></td>
</tr>
<tr>
<td><code>sed '/jan/s/-1/-5/'</code></td>
<td>Change the first <code>-1</code> to <code>-5</code> in all lines containing <code>jan</code></td>
</tr>
<tr>
<td><code>sed 's/...//' data</code></td>
<td>Delete the first three characters from each line of <code>data</code></td>
</tr>
<tr>
<td><code>sed 's/...$//' data</code></td>
<td>Delete the last 3 characters from each line of <code>data</code></td>
</tr>
<tr>
<td><code>sed -n '1' text</code></td>
<td>Print all lines from <code>text</code>, showing non-printing characters as <code>\nn</code> (where <code>nn</code> is the octal value of the character), and tab characters as <code>\t</code></td>
</tr>
</tbody>
</table>
The `tr` filter is used to translate characters from standard input. The general form of the command is:

```
tr from-chars to-chars
```

where `from-chars` and `to-chars` are one or more characters or a set of characters. Any character in `from-chars` encountered on the input will be translated into the corresponding character in `to-chars`. The result of the translation is written to standard output.

In its simplest form, `tr` can be used to translate one character into another. Recall the file `intro` from earlier in this chapter:

```
$ cat intro
The UNIX operating system was pioneered by Ken
Thompson and Dennis Ritchie at Bell Laboratories
in the late 1960s. One of the primary goals in
the design of the UNIX system was to create an
environment that promoted efficient program
development.
$
```

The following shows how `tr` can be used to translate all letter e's to x's:

```
$ tr e x < intro
Thx UNIX opaxrring systxm was pionxxrx by Kxn
Thompson and Dnxxis Rtxix at Bxxl Laboratorixs
in thx lax 1960s. Omx of thx primary goals in
thx dxsign of thx UNIX systxm was to crxatx an
xnvironmxnt that promotxd xfficixnt program
dvxxlogmxnt.
$
```

The input to `tr` must be redirected from the file `intro` because `tr` always expects its input to come from standard input. The results of the translation are written to standard output, leaving the original file untouched. Showing a more practical example, recall the pipeline that you used to extract the usernames and home directories of everyone on the system:

```
$ cut -d: -f1,6 /etc/passwd
root:/
cron:/
bin:/
uucp:/usr/spool/uucp
asg:/
steve:/users/steve
other:/
$
```

You can translate the colons into tab characters to produce a more readable output simply by tacking an appropriate `tr` command to the end of the pipeline:
Enclosed between the single quotes is a tab character (even though you can’t see it—just take our word for it). It must be enclosed in quotes to keep it from being parsed and discarded by the shell as extraneous whitespace.

Working with characters that aren’t printable? The octal representation of a character can be given to `tr` in the format

```
\nnn
```

where `nnn` is the octal value of the character. This isn’t used too often, but can be handy to remember.

For example, the octal value of the tab character is 11, so another way to accomplish the colon-to-tab transformation is to use the `tr` command

```
tr : '\11'
```

Table 3.3 lists characters that you’ll often want to specify in octal format.

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bell</td>
<td>7</td>
</tr>
<tr>
<td>Backspace</td>
<td>10</td>
</tr>
<tr>
<td>Tab</td>
<td>11</td>
</tr>
<tr>
<td>Newline</td>
<td>12</td>
</tr>
<tr>
<td>Newline</td>
<td>12</td>
</tr>
<tr>
<td>Formfeed</td>
<td>14</td>
</tr>
<tr>
<td>Carriage Return</td>
<td>15</td>
</tr>
<tr>
<td>Escape</td>
<td>33</td>
</tr>
</tbody>
</table>

In the following example, `tr` takes the output from `date` and translates all spaces into newline characters. The net result is that each field of output appears on a different line:

```
$ date | tr ' ' '\12'  Translate spaces to newlines
Sun
```
tr can also translate ranges of characters. For example, the following shows how to translate all lowercase letters in intro to their uppercase equivalents:

```
$ tr '[a-z]' '[A-Z]' < intro
THE UNIX OPERATING SYSTEM WAS PIONEERED BY KEN
THOMPSON AND DENNIS RITCHIE AT BELL LABORATORIES
IN THE LATE 1960S. ONE OF THE PRIMARY GOALS IN
THE DESIGN OF THE UNIX SYSTEM WAS TO CREATE AN
ENVIRONMENT THAT PROMOTED EFFICIENT PROGRAM
DEVELOPMENT.
```

The character ranges [a-z] and [A-Z] are enclosed in quotes to keep the shell from interpreting the pattern. Try the command without the quotes and you’ll quickly see that the result isn’t quite what you seek.

By reversing the two arguments to tr, you can use the command to translate all uppercase letters to lowercase:

```
$ tr '[A-Z]' '[a-z]' < intro
the unix operating system was pioneered by ken
thompson and dennis ritchie at bell laboratories
in the late 1960s. one of the primary goals in
the design of the unix system was to create an
environment that promoted efficient program
development.
```

For a more interesting example, try to guess what this tr invocation accomplishes:

```
tr '[a-zA-Z]' '[A-Za-z]'
```

Figured it out? This turns uppercase letters into lowercase, and lowercase letters into uppercase.

**The -s Option**

You can use the -s option to “squeeze” out multiple consecutive occurrences of characters in to-chars. In other words, if more than one consecutive occurrence of a character specified in to-chars occurs after the translation is made, the characters will be replaced by a single character.

For example, the following command translates all colons into tab characters, replacing multiple tabs with single tabs:

```
tr -s ':' '	'
```
So one colon or several consecutive colons on the input will be replaced by a *single* tab character on the output.

Note that '\t' can work in many instances instead of '\\t\t', so be sure to try that if you want things to be a bit more readable!

Suppose that you have a file called *lotsaspaces* that has contents as shown:

```bash
$ cat lotsaspaces
This is an example of a file that contains a lot of blank spaces.
$
```

You can use `tr` to squeeze out the multiple spaces by using the `-s` option and by specifying a single space character as the first and second argument:

```bash
$ tr -s ' ' ' ' < lotsaspaces
This is an example of a file that contains a lot of blank spaces.
$
```

This `tr` command in effect says, “translate occurrences of space with another space, replacing multiple spaces in the output with a single space.”

**The \(-d\) Option**

`tr` can also be used to delete individual characters from the input stream. The format of `tr` in this case is

```
tr -d from-chars
```

where any character listed in `from-chars` will be deleted from standard input. In the following example, `tr` is used to delete all spaces from the file `intro`:

```bash
$ tr -d ' ' < intro
TheUNIXopersingSystemwaspioneeredbyKenThompsonandDennisRitchieatBellLaboratories
inthe1960s.0neoftheprimarygoalsin
thedesigntoftheUNIXSystemwascreatean
evironmentthatpromotedefficientprogram
development.
$
```

You probably realize that you could have also used `sed` to achieve the same results:

```bash
$ sed 's/ //g' intro
TheUNIXopersingSystemwaspioneeredbyKenThompsonandDennisRitchieatBellLaboratories
inthe1960s.0neoftheprimarygoalsin
thedesigntoftheUNIXSystemwascreatean
evironmentthatpromotedefficientprogram
```
This is not atypical for the Unix system; there’s almost always more than one approach to solving a particular problem. In the case we just saw, either approach is satisfactory (that is, \texttt{tr} or \texttt{sed}), but \texttt{tr} is probably a better choice because it is a much smaller program and likely to execute faster.

Table 3.4 summarizes how to use \texttt{tr} for translating and deleting characters. Bear in mind that \texttt{tr} works only on single characters. So if you need to translate anything longer than a single character (say all occurrences of \texttt{unix} to \texttt{UNIX}), you have to use a different program, such as \texttt{sed}, instead.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\texttt{tr Command} & \texttt{Description} \\
\hline
\texttt{tr 'X' 'x'} & Translate all capital X’s to small x’s. \\
\hline
\texttt{tr '()' '{}'} & Translate all open parentheses to open braces, all closed parentheses to closed braces \\
\hline
\texttt{tr '[a-z]' '[A-Z]' } & Translate all lowercase letters to uppercase \\
\hline
\texttt{tr '[A-Z]' '[N-ZA-M]' } & Translate uppercase letters A–M to N–Z, and N–Z to A–M, respectively \\
\hline
\texttt{tr '    ' ' ' } & Translate all tabs (character in first pair of quotes) to spaces \\
\hline
\texttt{tr -s ' ' ' } & Translate multiple spaces to single spaces \\
\hline
\texttt{tr -d '\14'} & Delete all formfeed (octal 14) characters \\
\hline
\texttt{tr -d '0-9'} & Delete all digits \\
\hline
\end{tabular}
\caption{\texttt{tr} Examples}
\end{table}

grep

grep allows you to search one or more files for a pattern you specify. The general format of this command is

\texttt{grep pattern files}

Every line of each file that contains \texttt{pattern} is displayed at the terminal. If more than one file is specified to \texttt{grep}, each line is also preceded by the name of the file, thus enabling you to identify the particular file that the pattern was found in.

Let’s say that you want to find every occurrence of the word \texttt{shell} in the file \texttt{ed.cmd}:

\$ \texttt{grep shell ed.cmd}

files, and is independent of the shell.
to the shell, just type in a q.
\$

This output indicates that two lines in the file \texttt{ed.cmd} contain the word \texttt{shell}.
If the pattern does not exist in the specified file(s), the `grep` command simply displays nothing:

```bash
$ grep cracker ed.cmd

You saw in the section on `sed` how you could print all lines containing the string `UNIX` from the file `intro` with the command

```bash
sed -n '/UNIX/p' intro
```

But you could also use the following `grep` command to achieve the same result:

```bash
grep UNIX intro
```

Recall the `phonebook` file from before:

```bash
$ cat phonebook
Alice Chebba  973-555-2015
Barbara Swingle 201-555-9257
Jeff Goldberg  201-555-3378
Liz Stachiw    212-555-2298
Susan Goldberg 201-555-7776
Tony Iannino   973-555-1295

$ 
```

When you need to look up a particular phone number, the `grep` command comes in handy:

```bash
$ grep Susan phonebook
Susan Goldberg  201-555-7776

$ 
```

The `grep` command is particularly useful when you have a lot of files and you want to find out which ones contain certain words or phrases. The following example shows how the `grep` command can be used to search for the word `shell` in all files in the current directory:

```bash
$ grep shell *
```

As noted, when more than one file is specified to `grep`, each output line is preceded by the name of the file containing that line.

As with expressions for `sed` and patterns for `tr`, it's a good idea to enclose your `grep` pattern inside a pair of `single` quotes to “protect” it from the shell. Here’s an example of what can happen if you don’t: say you want to find all the lines containing asterisks inside the file `stars`; typing

```bash
grep * stars
```
doesn’t work as you’d hope because the shell sees the asterisk and automatically substitutes the names of all the files in your current directory!

```bash
$ ls
circles
polka.dots
squares
stars
stripes
$ grep * stars
```

In this case, the shell took the asterisk and substituted the list of files in your current directory. Then it started execution of `grep`, which took the first argument (`circles`) and tried to find it in the files specified by the remaining arguments, as shown in Figure 3.1.

```
grep
  arguments
  circles
  polka.dots
  squares
  stars
  stripes
  stars
```

Figure 3.1  grep * stars

Enclosing the asterisk in quotes, however, blocks it from being parsed and interpreted by the shell:

```bash
$ grep "*" stars
```

The asterisk (*) is a special character that
```
5 * 4 = 20
```

The quotes told the shell to leave the enclosed characters alone. It then started execution of `grep`, passing it the two arguments `*` (*without* the surrounding quotes; the shell removes them in the process) and `stars` (see Figure 3.2).

```
grep
  arguments
  *
  stars
```

Figure 3.2  grep ' *' stars
There are characters other than * that have a special meaning to the shell and must be quoted when used in a pattern. The whole topic of how quotes are handled by the shell is admittedly tricky; an entire chapter—Chapter 5—is devoted to it.

grep takes its input from standard input if no filename is specified. So you can use grep as part of a pipe to scan through the output of a command for lines that match a specific pattern. Suppose that you want to find out whether the user jim is logged in. You can use grep to search through who's output:

```
$ who | grep jim
jim        tty16             Feb 20 10:25
```

Note that by not specifying a file to search, grep automatically scans standard input. Naturally, if the user jim were not logged in, you would get a new command prompt without any preceding output:

```
$ who | grep jim
```

### Regular Expressions and grep

Let's take another look at the intro file:

```
$ cat intro
The UNIX operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in the design of the UNIX system was to create an environment that promoted efficient program development.
```

grep allows you to specify your pattern using regular expressions as in ed. Given this information, it means that you can specify the pattern `[tT]he` to have grep search for either a lower- or uppercase T followed by the characters he.

Here's how to use grep to list all the lines containing the characters the or The:

```
$ grep '^[tT]he' intro
The UNIX operating system was pioneered by Ken in the late 1960s. One of the primary goals in the design of the UNIX system was to create an
```

A smarter alternative might be to utilize the -i option to grep which makes patterns case insensitive. That is, the command

```
grep -i 'the' intro
```
Chapter 3  Tools of the Trade

tells grep to ignore the difference between upper and lowercase when matching the pattern against the lines in intro. Therefore, lines containing the or The will be printed, as will lines containing THE, The, the, and so on.

Table 3.5 shows other types of regular expressions that you can specify to grep and the types of patterns they’ll match.

Table 3.5  Some grep Examples

<table>
<thead>
<tr>
<th>Command</th>
<th>Prints</th>
</tr>
</thead>
<tbody>
<tr>
<td>grep '[A-Z]' list</td>
<td>Lines from list containing a capital letter</td>
</tr>
<tr>
<td>grep '[0-9]' data</td>
<td>Lines from data containing a digit</td>
</tr>
<tr>
<td>grep '[A-Z][0-9]' list</td>
<td>Lines from list containing five-character patterns that start with a capital letter and end with a digit</td>
</tr>
<tr>
<td>grep '.pic$' filelist</td>
<td>Lines from filelist that end with .pic</td>
</tr>
</tbody>
</table>

The -v Option

Sometimes you’re interested not in finding the lines that contain a specified pattern, but those that don’t. That’s what the -v option is for with grep: to reverse the logic of the matching task.

In the next example, grep is used to find all the lines in intro that don’t contain the pattern UNIX.

$ grep -v 'UNIX' intro  
Print all lines that don’t contain UNIX

Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s. One of the primary goals in environment that promoted efficient program development.

$ The -l Option

At times, you may not want to see the actual lines that match a pattern but just seek the names of the files that contain the pattern. For example, suppose that you have a set of C programs in your current directory (by convention, these filenames end with the filename suffix .c), and you want to know which use a variable called Move_history. Here’s one way of finding the answer:

$ grep 'Move_history' *.c  
Find Move_history in all C source files

exec.c: MOVE Move_history[200] = {0};
exec.c: cpymove(&Move_history[Number_half_moves -1],
exec.c: undo_move(&Move_history[Number_half_moves-1]),
exec.c: cpymove(&last_move,&Move_history[Number_half_moves-1]);
exec.c: convert_move(&Move_history[Number_half_moves-1]),
exec.c: convert_move(&Move_history[1-1]),
exec.c: convert_move(&Move_history[Number_half_moves-1]),
makemove.c: IMPORT MOVE Move_history[];
makemove.c: if ( Move_history[j].from != BOOK (i,j,from) OR
makemove.c: Move_history[j].to != BOOK (i,j,to) )
testch.c:GLOBAL MOVE Move_history[100] = {0};
testch.c: Move_history[Number_half_moves-1].from = move.from;
testch.c: Move_history[Number_half_moves-1].to = move.to;
$

Sifting through the preceding output, you discover that three files—exec.c, makemove.c, and
testch.c—use the variable.

Add the -l option to grep and you instead get a list of files that contain the specified pattern,
not the matching lines from the files:

$ grep -l 'Move_history' *.c
List the files that contain Move_history
exec.c
makemove.c
testch.c
$

Because grep conveniently lists the files one per line, you can pipe the output from grep -l
into wc to count the number of files that contain a particular pattern:

$ grep -l 'Move_history' *.c | wc -l
3
$

The preceding command shows that precisely three C program files reference the variable
Move_history. Now, just to make sure you’re paying attention, what are you counting if you
use grep without the -l option and pipe the output to wc -l?

The -n Option

If the -n option is used with grep, each line from the file that matches the specified pattern
is preceded by its corresponding line number. From previous examples, you saw that the file
testch.c was one of the three files that referenced the variable Move_history; the following
shows how you can pinpoint the precise lines in the file that reference the variable:

$ grep -n 'Move_history' testch.c
Precede matches with line numbers
13:GLOBAL MOVE Move_history[100] = {0};
197: Move_history[Number_half_moves-1].from = move.from;
198: Move_history[Number_half_moves-1].to = move.to;
$

As you can see, Move_history is used on lines 13, 197, and 198 in testch.c.

For Unix experts, grep is one of the most commonly used programs because of its flexibility
and sophistication with pattern matching. It’s one well worth studying.
sort

At its most basic, the `sort` command is really easy to understand: give it lines of input and it'll sort them alphabetically, with the result appearing as its output:

```
$ sort names
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
Tony
$
```

By default, `sort` takes each line of the specified input file and sorts it into ascending order.

Special characters are sorted according to the internal encoding of the characters. For example, the space character is represented internally as the number 32, and the double quote as the number 34. This means that the former would be sorted before the latter. Particularly for other languages and locales the sorting order can vary, so although you are generally assured that `sort` will perform as expected on alphanumeric input, the ordering of foreign language characters, punctuation, and other special characters is not always what you might expect.

`sort` has many options that provide more flexibility in performing your sort. We'll just describe a few of the options here.

The `-u` Option

The `-u` option tells `sort` to eliminate duplicate lines from the output.

```
$ sort -u names
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
$
```

Here you see that the duplicate line that contained `Tony` was eliminated from the output. A lot of old-school Unix people accomplish the same thing by using the separate program `uniq`, so if you read system shell scripts you'll often see sequences like `sort | uniq`. Those can be replaced with `sort -u`!
The `-r` Option

Use the `-r` option to reverse the order of the sort:

```bash
$ sort -r names  
Tony
Tony
Ralph
Lucy
Fred
Emanuel
Charlie
$
```

The `-o` Option

By default, `sort` writes the sorted data to standard output. To have it go into a file, you can use output redirection:

```bash
$ sort names > sorted_names
$
```

Alternatively, you can use the `-o` option to specify the output file. Simply list the name of the output file right after the `-o`:

```bash
$ sort names -o sorted_names
$
```

This sorts `names` and writes the results to `sorted_names`.

What’s the value of the `-o` option? Frequently, you want to sort the lines in a file and have the sorted data replace the original. But typing

```bash
$ sort names > names
$
```

won’t work—it ends up wiping out the `names` file! However, with the `-o` option, it is okay to specify the same name for the output file as the input file:

```bash
$ sort names -o names
$ cat names
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
Tony
$
```
Tip

Be careful if your filter or process is going to replace your original input file and make sure that
it’s all working as you expect prior to having the data overwritten. Unix is good at a lot of things,
but there’s no unremove command to recover lost data or lost files.

The -n Option

Suppose that you have a file containing pairs of \((x, y)\) data points as shown:

```
$ cat data
  5    27
  2    12
  3    33
  23   2
 -5   11
 15    6
 14    -9
$
```

And suppose that you want to feed this data into a plotting program called plotdata, but that
the program requires that the incoming data pairs be sorted in increasing value of \(x\) (the first
value on each line).

The -n option to sort specifies that the first field on the line is to be considered a number,
and the data is to be sorted arithmetically. Compare the output of sort used without
the -n option and then with it:

```
$ sort data
 -5   11
14    -9
15    6
 2    12
 23   2
 3    33
 5    27
$
```

```
$ sort -n data  Sort arithmetically
 -5   11
 2    12
 3    33
 5    27
14    -9
15    6
 23   2
$
```

Skipping Fields

If you had to sort your data file by the y value—that is, the second number in each line—you could tell `sort` to start with the second field by using the option

```
-k2n
```

instead of `-n`. The `-k2` says to skip the first field and start the sort analysis with the second field of each line. Similarly, `-k5n` would mean to start with the fifth field on each line and then sort the data numerically.

```
$ sort -k2n data
Start with the second field in the sort
  14   -9
  23    2
  15    6
  -5    11
   2    12
   5    27
   3    33
$
```

Fields are delimited by space or tab characters by default. If a different delimiter is to be used, the `-t` option must be used.

The `-t` Option

As mentioned, if you skip over fields, `sort` assumes that the fields are delimited by space or tab characters. The `-t` option can indicate otherwise. In this case, the character that follows the `-t` is taken as the delimiter character.

Consider the sample password file again:

```
$ cat /etc/passwd
root:*:0:0:The super User::/usr/bin/ksh
steve:*:203:100::/users/steve:/usr/bin/ksh
bin:*:3:3:The owner of system files::
cron:*:1:1:Cron Daemon for periodic tasks::
george:*:75:75::/users/george:/usr/lib/rsh
pat:*:300:300::/users/pat:/usr/bin/ksh
uucp:nc823ciSiLiZM:5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico
asg:*:6:6:The Owner of Assignable Devices::
sysinfo:*:10:10:Access to System Information::/usr/bin/sh
mail:*:301:301::/usr/mail:
$
```

If you wanted to sort this file by username (the first field on each line), you could just issue the command

```
sort /etc/passwd
```
To sort the file instead by the third colon-delimited field (which contains what is known as your user ID), you would want an arithmetic sort, starting with the third field (-k3), and specifying the colon character as the field delimiter (-t:):

```
$ sort -k3n -t: /etc/passwd  # Sort by user id
```

Here we've bolded the third field of each line so that you can easily verify that the file was sorted correctly by user ID.

**Other Options**

Other options to `sort` enable you to skip characters within a field, specify the field to end the sort on, merge sorted input files, and sort in “dictionary order” (only letters, numbers, and spaces are used for the comparison). For more details on these options, look under `sort` in your *Unix User's Manual*.

**uniq**

The `uniq` command is useful when you need to find or remove duplicate lines in a file. The basic format of the command is

```
uniq in_file out_file
```

In this format, `uniq` copies `in_file` to `out_file`, removing any duplicate lines in the process. `uniq`'s definition of duplicated lines is *consecutive lines that match exactly*.

If `out_file` is not specified, the results will be written to standard output. If `in_file` is also not specified, `uniq` acts as a filter and reads its input from standard input.

Here are some examples to see how `uniq` works. Suppose that you have a file called `names` with contents as shown:

```
$ cat names
Charlie
Tony
Emanuel
Lucy
```

```
$ uniq names
Charlie
Tony
Emanuel
Lucy
```
You can see that the name Tony appears twice in the file. You can use `uniq` to remove such duplicate entries:

```
$ uniq names
Print unique lines
Charlie
Tony
Emanuel
Lucy
Ralph
Fred
Tony
$
```

Oops! Tony still appears twice in the preceding output because the multiple occurrences are not consecutive in the file, and thus `uniq`'s definition of duplicate is not satisfied. To remedy this situation, `sort` is often used to get the duplicate lines adjacent to each other, as mentioned earlier in the chapter. The result of the sort is then run through `uniq`:

```
$ sort names | uniq
Charlie
Emanuel
Fred
Lucy
Ralph
Tony
$
```

The `sort` moves the two Tony lines together, and then `uniq` filters out the duplicate line (but recall that `sort` with the `-u` option performs precisely this function).

### The `-d` Option

Frequently, you'll be interested in finding just the duplicate entries in a file. The `-d` option to `uniq` can be used for such purposes: It tells `uniq` to write only the duplicated lines to `out_file` (or standard output). Such lines are written just once, no matter how many consecutive occurrences there are.

```
$ sort names | uniq -d
List duplicate lines
Tony
$
```

As a more practical example, let's return to our `/etc/passwd` file. This file contains information about each user on the system. It's conceivable that over the course of adding and removing users from this file that perhaps the same username has been inadvertently entered
more than once. You can easily find such duplicate entries by first sorting /etc/passwd and piping the results into uniq -d as done previously:

```
$ sort /etc/passwd | uniq -d    Find duplicate entries in /etc/passwd
```

There are no duplicate full line /etc/passwd entries. But you really want to find duplicate entries for the username field, so you only want to look at the first field from each line (recall that the leading characters of each line of /etc/passwd up to the colon are the username). This can't be done directly through an option to uniq, but can be accomplished by using cut to extract the username from each line of the password file before sending it to uniq.

```
$ sort /etc/passwd | cut -f1 -d: | uniq -d    Find duplicates
cem
harry
```

It turns out that there are multiple entries in /etc/passwd for cem and harry. If you wanted more information on the particular entries, you could now grep them from /etc/passwd:

```
$ grep -n 'cem' /etc/passwd
20:cem:*:91:91::/users/cem:
166:cem:*:91:91::/users/cem:
$ grep -n 'harry' /etc/passwd
29:harry:*:103:103:Harry Johnson:/users/harry:
79:harry:*:90:90:Harry Johnson:/users/harry:
```

The -n option was used to find out where the duplicate entries occur. In the case of cem, there are two entries on lines 20 and 166; in harry's case, the two entries are on lines 29 and 79.

**Other Options**

The -c option to uniq adds an occurrence count, which can be tremendously useful in scripts:

```
$ sort names | uniq -c    Count line occurrences
  1 Charlie
  1 Emanuel
  1 Fred
  1 Lucy
  1 Ralph
  2 Tony
```

One common use of uniq -c is to figure out the most common words in a data file, easily done with a command like:

```
tr '[A-Z]' '[a-z]' datafile | sort | uniq -c | head
```
Two other options that we don’t have space to describe more fully let you tell uniq to ignore leading characters/fields on a line. For more information, consult the man page for your particular implementation of uniq with the command man uniq.

We would be remiss if we neglected to mention the programs awk and perl, which can be useful when writing shell programs too. They are both big, complicated programming environments unto themselves, however, so we’re going to encourage you to check out Awk—A Pattern Scanning and Processing Language, by Aho, et al., in the Unix Programmer’s Manual, Volume II for a description of awk, and Learning Perl and Programming Perl, both from O’Reilly and Associates, offering a good tutorial and reference on the language, respectively.
This page intentionally left blank
Symbols

& (ampersand)
  && construct, 161–162
  background execution of loops, 177
  command sequences and, 322

* (asterisk)
  with case statement, 155
  filename substitution, 24–25, 47, 331
  pattern matching, 57–59, 63, 242, 336

` (back quote), 114–115

\ (backslash)
  escaping characters with, 111–112
  inside double quotes, 112–114
  line continuation, 112
  overview, 322

[ ] (brackets), 139–140

^ (caret), 53, 63

: (colon)
  : (null) command, 160–161, 334
  in directories, 218

((.) construct, 311

(. ) construct, 47, 231–234, 332

\(\) construct, 61–63, 66

\(\) construct, 59–61, 64

#! construct, 289–290

| | construct, 161–162

$(.) construct, 115–118

[!] construct, 242
[.] construct, 55–57, 242, 336
{ ; : } construct, 231–234, 332
> & construct, 261–262
>&& construct, 262
< & construct, 262
$ (dollar sign)
  command prompt, 43
  parameter substitution
    ${parameter}$, 239–240
    ${parameter:+value}$, 242
    ${parameter:=value}$, 241
    ${parameter:=value}$, 240
    ${parameter:=value}$, 241–242
  pattern matching, 53–54, 63
  variable substitution, 98–100
" (double quotes)
  backslash (\) in, 112–114
  grouping character sequences with, 109–111
- (hyphen)
  command options, 8
  job control, 315
  printf format specification modifier, 206
< (left arrow), 48–49, 331–332
! (logical negation) operator, 143, 322
$(( )) operator, 103
% (percent sign)
  % format specification character, 203
%b format specification character, 203
%c format specification character, 203
%e format specification character, 203
%o format specification character, 203
%s format specification character, 203
%s format specification character, 203
%u format specification character, 203
%X format specification character, 203
%x format specification character, 203
  job control, 315
  pattern matching, 242–243
  with printf command, 202
. (period)
  dot (.) command, 227–230, 334–335
  pattern matching, 51–53, 63
| (pipe symbol)
  || construct, 161–162
  case command, 159–160
  loops, 178–179
  pipeline process, 33–34, 49, 321, 322
+ (plus sign)
  job control, 315
  pattern matching, 63
  printf format specification modifier, 206
# (pound sign)
  comments, 96
  pattern matching, 243
  printf format specification modifier, 206
? (question mark)
  filename substitution, 25–27, 47, 331
  pattern matching, 242, 336
>> (redirection append) characters, 31–32, 48–49
<< (redirection append) characters
  shell archive creation, 264–267
  syntax, 262–264
> (right arrow), 30–32, 48–49, 331–332
; (semicolon), 36, 179–180, 321
' (single quotes), 105–108
~ (tilde) substitution, 318–319, 329
_ (underscore), 322
$? $! variable, 323
$! variable, 257–258, 323
bases, numbers in different bases  365

$- variable, 323
#$ variable, 122–123, 323
$$ variable, 198–199, 323
$+ variable, 123–124, 166, 323
$@ variable, 166–167, 323
$0 variable, 245, 323
${(n)} variable, 128
${(variable)} construct, 98–100, 102–103

A

-a (logical AND) operator, 143–144
access modes, 16–17
accessing command history.
  See command history
add program, 125–127, 277
addi program, 200
alias command, 307–309, 335
aliases
  defining, 307–309
  removing, 309
allexport shell mode, 350
alternative format for test command, 139–140
ampersand (&)
  && construct, 161–162
  background execution of loops, 177
  command sequences and, 322
archives, creating, 264–267
args program, 122–123
arguments
  definition of, 321
  passing
    $# variable, 122–123
    $ variable, 123–124
    $[n] special variable, 128
    phonebook file example, 124–128

B

%b format specification character, 203
back quote (‘), 114–115
background execution
  commands, 36–37
  jobs, 316–317
  loops, 177
backslash (\)
  escaping characters with, 111–112
  inside double quotes, 112–114
  line continuation, 112
  overview, 322
bases, numbers in different bases, 305–306
Bash shell. See also nonstandard shell features
- compatibility summary, 319–320
- history of, 289
- Web documentation, 360

beginning of line, matching, 53
Bell Laboratories, 1
bg command, 316–317, 335
blocks of storage, 16
body of loops, 164
books, recommended, 360–361
Bourne, Stephen, 1
brackets ([ ]), 139–140
break command, 174–176, 336
breaking out of loops, 174–176

C

C compiler, 360
% format specification character, 203
caret (^), 53, 63
case command
debugging, 157–159
overview, 336–337
pattern-matching characters, 155–156
pipe symbol (|), 159–160
syntax, 153–154
cat command, 7
cd command, 12–15, 317–318, 337
cdh function, 312–314
CDPATH variable, 323, 337
cdtest program, 225
change program, 281–283
changing
directories, 12–15, 337
group id (GID), 347–348
phonebook file entries, 281–283
character sets, matching, 55–57

characters. See also text
- ASCII characters, octal values of, 75
- character sequences
  - double quotes ("), 109–111
  - single quotes (‘), 105–108
- cutting, 64–68
deleting from input stream, 77–78
echoing, 6
escaping, 111–112
in filenames
  - allowed characters, 6
  - special characters, 28
format specification characters (printf), 202–205
pattern matching
  - any character, 51–53
  - beginning of line, 53
  - character sets, 55–57
  - end of line, 53–54
  - filename substitution, 25–27
  - matched characters, saving, 61–63
  - overview, 155–156
  - parameter substitution, 242–244
  - precise number of characters, 59–61
  - zero or more characters, 57–59
quote characters
  - backslash (\), 111–114
  - double quotes ("), 109–111
  - single quotes (‘), 105–108
translating from standard input, 74–77

child processes, 257
classes, else, 145–147, 345–346
closing standard output, 262
colon (:\n
: (null) command, 334
in directories, 218
for command
  $* variable, 166
  $@ variable, 166–167
overview, 163–166, 342–343
without in element, 167–168
command cycle, 43
command files, 93–96
command history
  accessing
    emacs line edit mode, 294–296
    fc command, 301, 326
    history command, 294–296, 300–301
    quoting, 329
    r command, 301–303
    vi line edit mode, 294–296, 326–329
controlling size of, 292
editing commands in, 341
command line. See commands
Command not found error, 94
command prompt, 43
command substitution
  $(.) construct, 115–118
back quote ("), 114–115
definition of, 112
expr command, 119–120
command-line editing
command history
  accessing with vi, 294–296
controlling size of, 292
emacs line edit mode
  command history, accessing, 296–298
overview, 296–298
overview, 291
vi line edit mode
  command history, accessing, 294–296
overview, 292–294
commands
  $(.) construct, 332
  [ ]; ] construct, 332
  alias, 307–309, 335
arguments, passing
  $# variable, 122–123
  $* variable, 123–124
  ${[n]} special variable, 128
overview, 343–344
phonebook file example, 124–128
positioned parameters, 121–122
shift command, 128–129
awk, 91
bg, 316–317, 335
break, 174–176, 336
case
debbuging, 157–159
overview, 336–337
pattern-matching characters, 155–156
pipe symbol (|), 159–160
syntax, 153–154
cat, 7
cd, 12–15, 317–318, 337
command cycle, 43
command history
  accessing, 326–329
  editing commands in, 341
command re-entry, 326
command summary, 37–38
continue, 176–177, 338
cp, 8, 18–19
cut
- d option, 66–68
- f option, 66–68
overview, 64–66
date, 5, 95–96, 237
dot (.), 334–335
echo
escape characters, 187–188
overview, 6, 338–339
emacs line edit commands, 299–300
entering, 43–44
eval, 255–257, 339
exec, 230–231, 262, 339
exit, 147–148, 340
export, 340–341
expr, 119–120
false, 341
fc, 301, 326–201.2304, 341
fg, 316–317, 342
for
- * variable, 166
- @ variable, 166–167
overview, 163–166, 342–343
without in element, 167–168
format of, 321–322
getopts, 180–184, 343–344
grep
- l option, 82–83
- n option, 83
overview, 78–81
regular expressions, 81–82
-v option, 82
grouping, 231–234
hash, 344
history, 294–296, 300–301
if. See if statement
info, 359
jobs, 315, 347
kill, 315, 347
ln, 20–23
ls, 7, 15–17
man, 359
mkdir, 17–18
multiple commands on same line, 36
mv, 8–9, 19
newgrp, 347–348
null (.), 160–161, 334
od, 251
options, 8
paste
- d option, 69–70
overview, 68–69
-s option, 70
perl, 91
printf
example, 206–207
format specification characters,
202–205
format specification modifiers,
205–206
syntax, 202
printing information about, 356
ps, 37
pwd, 12, 95–96, 348
quoting, 329
r, 301–303
read
exit status, 199–202
menu-driven phone program (rolo),
193–199
mycp program, 185–193
overview, 348–349
syntax, 185
readonly, 349
compatibility of shells

readyonly, 254
return, 271, 349
returning information about, 271
rm, 9
rmdir, 22–23
scanning twice before executing, 255–257
sed
d command, 73
dexamples, 73
-n option, 72
syntax, 70–72
sending to background, 36–37
set
-- option, 248–250
IFS variable, 251–254
monitor option, 331–332
with no arguments, 247
overview, 239, 321, 350–351
positional parameters, reassigning, 247–248
-x option, 246
shift, 128–129, 352
skipping in loops, 176–177
sort
-k2n option, 87
-n option, 86
-o option, 85
other options, 88
overview, 84
-r option, 85
-t option, 87–88
-u option, 84
substitution
$(.) construct, 115–118
back quote (‘), 114–115
definition of, 112
expr command, 119–120
test, 135, 352–354
tilde substitution, 318–319, 329
times, 354, 355–356
tr
-d option, 77–78
examples, 78
octal values of ASCII characters, 75
overview, 74–76
-s option, 76–77
trap
execution with no arguments, 259–260
ignored signals, 260
overview, 258–259
signal numbers, 258
trap reset, 261
true, 356
type, 271
typing on one line, 179–180
umask, 356
unalias, 309, 356
uniq
-c option, 90
-d option, 89–90
overview, 88–89
unset, 254, 271, 357
until, 170–174, 357
vi line edit commands, 296, 326–329
wait, 257, 358
wc, 7, 95–96
while, 168–170
who, 5–6

comments, 96, 322
compatibility of shells, 319–320
conditional statements. See also loops
& construct, 161–162
|| construct, 161–162
if
case command, 153–160
e1f construct, 148–151
e1se construct, 145–147
exit command, 147–148
exit status, 131–135
null command (:), 160–161
pipe symbol (), 159–160
syntax, 131
testing conditions in, 131–144
nesting, 148–149
testing conditions in
alternative format for test, 139–140
file operators, 142–143
integer operators, 140–142
logical AND operator (-a), 143–144
logical negation operator (!), 143
logical OR operator (-o), 144
overview, 135
string operators, 135–139
contents of files, displaying, 7
continuation of lines, 112
continue command, 176–177, 338
Coordinated Universal Time, 237
copying files
to another directory, 18–19
mycp program
echo escape characters, 187–188
final code listing, 190–193
initial code listing, 185–187
revised code listing, 188–190
into new file, 8
counting words in files, 7
cp command, 8, 18–19
creating
aliases, 307–309
directories, 17–18
functions, 268
pointers to variables, 257
shell archives, 264–267
c type program, 155–156, 158–159
current directory, 225–226
cut command
-d option, 66–68
-f option, 66–68
overview, 64–66
cygwin, 360
d command, 73
-d file operator, 142–143
%d format specification character, 203
dangling symbolic links, 203
data. See also l/O (input/output) redirection
extracting
cut command syntax, 64–66
delimiter characters, 66–68
fields, 66–68
printing
command information, 356
date/time, 5, 95–96
formatted output, 202–207
list of active jobs, 347
to working directory, 348
reading
exit status, 199–202
menu-driven phone program (rolo)
example, 193–199
mycp program, 185–193
read command syntax, 185
data formatting (rolo program), 273–274
**duplicate entries, finding** 371

date command, 5, 95–96, 237

date/time, printing, 5, 95–96

db program, 227–229

debugging with -x, 157–159

defining. See creating
definitions (function)
creating, 268
removing, 271

deleting
aliases, 309
duplicate lines
sort command, 84
uniq command, 88–89
duplicates
files, 9
function definitions, 271
lines, 73
phonebook file entries, 127–128, 280–281

delimiter characters
cut command, 66–68
paste command, 69–70
sort command, 87–88
documentation
books, 360–361
comments, 96
here documents
shell archive creation, 264–267
syntax, 262–264
online documentation, 359
Web documentation, 360
dollar sign ($)
command prompt, 43
parameter substitution
${parameter}, 239–240
${parameter:+value}, 242
${parameter:=value}, 241
${parameter:=-value}, 240
${parameter:?value}, 241–242

dot (.) command, 227–230

double quotes ("")
backslash (\) in, 112–114

duplicate entries, finding, 89–90
duplicate lines, eliminating
   sort command, 84
   uniq command, 88–89

E
-e file operator, 142–143
echo command
   escape characters, 187–188
   overview, 6, 338–339
editing
   command-line editing
      command history, 292
      emacs line edit mode, 296–300
      overview, 291
      vi line edit mode, 292–296
   phonebook file entries, 281–283
editors
   stream editor (sed)
      command syntax, 70–72
      d command, 73
      examples, 73
      -n option, 72
      vi line edit mode, 326–329
elements of arrays, retrieving, 309–310
elif construct, 148–151
else clause, 145–147, 345–346
emacs line edit mode
   command history, accessing, 296–298
   overview, 296–298
enabling trace mode, 246
end of line, matching, 53–54
end-of-line character, 45
entering
   commands, 43–44
   passwords at login, 41
entries (phonebook file)
   adding, 125–127, 277
   displaying, 278–279
   editing, 281–283
   listing, 283–284
   looking up, 124–125, 278
   removing, 127–128, 280–281
ENV file, 290–291
ENV variable, 290–291, 323
environment
   current directory, 225–226
   exported variables, 211–216
   HOME variable, 217
   local variables, 209–210
   PATH variable, 217–224
   .profile file, 235–236
   PS1 variable, 216
   PS2 variable, 216
subshells
   () construct, 231–234
   { .; } construct, 231–234
   dot () command, 227–230
   exec command, 227–230
   overview, 210–211, 227
   passing variables to, 234–235
   TERM variable, 236–237
   TZ variable, 236–237
environment control, 49
-eq operator, 140–142
errexit shell mode, 350
errors
   Command not found, 94
   error messages, displaying, 245
   standard error, 35
escape characters (echo command), 187–188, 338
escaping characters, 111–112
/etc/passwd, 41
/etc/profile, 235–236
/etc/shadow, 41
eval command, 255–257, 339
exclamation mark (!), 322
exec command, 230–231, 262, 339
executable files, 94
execution
  background execution of loops, 177
  command execution
    command files, 93–96
    scanning command line twice before executing, 255–257
  in current shell, 227–230
  function execution, 268–269
  program execution, 45–47
  subshell execution, 332
exit command, 147–148, 340
EXIT signal, 258
exit status
  $? variable, 132–135
  definition of, 321
  non-zero values, 131
  overview, 131–132
  read command, 199–202
  zero values, 131
export command, 340–341
exported variables, 211–216, 332, 340–341
expr command, 119–120
expressions
  arithmetic expressions, 330
  regular expressions
    [.] construct, 55–57
    \(\).\) construct, 61–63
    \[\].\] construct, 59–61
    asterisk (*), 57–59
caret (^), 53
dollar sign ($), 53–54
grep command, 81–82
overview, 51
period (.), 51–53
summary table, 61–63
extracting data
cut command
  delimiter characters, 66–68
  fields, 66–68
  overview, 64–66
-f file operator, 142–143
false command, 341
fc command, 301, 326–Z01.2304, 341
FCEDIT variable, 323
fg command, 316–317, 342
fields
  cutting, 66–68
  skipping during sort, 87
file descriptors, 261
file operators, 142–143
filename substitution
  asterisk (*), 24–25
  overview, 47, 331
  POSIX shell, 331
  question mark (?), 25–27
  variables, 101–103
files
  command files, 93–96
  copying
    to another directory, 18–19
    mycp program, 185–193
  into new file, 8
  counting words in, 7
directory files, 6
displaying contents of, 7
duplicate entries, finding, 89–90
ENV, 290–291
executable files, 94
executing in current shell, 227–230
file descriptors, 261
file operators, 142–143
filename substitution
  * (asterisk), 24–25
  ? (question mark), 25–27
  asterisk (*), 24–25
  overview, 47, 331
  POSIX shell, 331
  question mark (?), 25–27
  variables, 101–103
filenames
  allowed characters, 6
  changing, 8–9
  spaces in, 27
  special characters, 28
linking, 20–23
listing, 7, 15–17
moving between directories, 19
ordinary files, 6
overview, 6–7
phonebook
  adding entries to, 125–127, 277
  displaying entries from, 278–279
  editing entries in, 281–283
  listing entries in, 283–284
  looking up entries in, 124–125, 278
  removing entries from, 127–128, 280–281
  .profile, 235–236
removing, 9
renaming, 8–9
rolosubs file, 264–266
searching with grep
  -l option, 82–83
  -n option, 83
  overview, 78–81
  regular expressions, 81–82
  -v option, 82
  sorting lines into, 85
  special files, 6
  temporary files, 198–199
filters, 35
finding. See pattern matching
foreground jobs
  bringing jobs to, 342
  stopping, 316–317
format specification (printf)
  characters, 202–205
  modifiers, 205–206
formatted output, printing
  example, 206–207
  format specification characters, 202–205
  format specification modifiers, 205–206
  printf command syntax, 202
Fox, Brian, 289
Free Software Foundation, 289, 360
fsf.org website, 360
functions
  automatically loaded functions, 303
cdh, 312–314
definitions
  creating, 268–271
  removing, 271
execution, 268–269
local variables, 303
overview, 333
terminating, 271
if statement

G
-ge operator, 140–142
getopts command, 180–184, 343–344
getty program. See shells
GID (group id), changing, 347–348
greetings program, 149–151, 159–160
grep command
-1 option, 82–83
-n option, 83
overview, 78–81
regular expressions, 81–82
-v option, 82
-group id (GID), changing, 347–348
grouping commands, 231–234
groups, 16–17
-gt operator, 140–142

H
handing signals with trap command
execution with no arguments, 259–260
ignored signals, 260
overview, 258–259
signal numbers, 258
trap reset, 261
hash command, 344
hash sign (#)
comments, 96
pattern matching, 243
printf format specification modifier, 206
here documents
shell archive creation, 264–267
syntax, 262–264
HISTFILE variable, 323
history command, 294–296, 300–301.
See also command history

history of Unix, 1
HISTSIZE variable, 326
home directory, 10–12, 217
HOME variable, 217, 217
HUP signal, 258
hyphen (-)
command options, 8
job control, 315
printf format specification modifier, 206

I
If statement
case command
debugging, 157–159
pattern-matching characters, 155–156
pipe symbol (|), 159–160
syntax, 153–154
elif construct, 148–151
else construct, 145–147
exit command, 147–148
exit status
$? variable, 132–135
non-zero values, 131
overview, 131–132
zero values, 131
nesting, 148–149
null command (\), 160–161
overview, 344–346
syntax, 131
testing conditions in
alternative format for test, 139–140
file operators, 142–143
integer operators, 140–142
logical AND operator (-a), 143–144
logical negation operator (!), 143
logical OR operator (-o), 144
overview, 135
parentheses, 144
string operators, 135–139
test command syntax, 135
IFS variable, 251–254, 323
ignoreeof shell mode, 351
ignoring signals, 260
infinite loops, breaking out of, 174–176
info command, 359
init program, 40–43
input redirection
< (left arrow), 331
exec command, 230–231, 262
in-line input redirection
shell archive creation, 264–267
syntax, 262–264
POSIX shell, 331
standard I/O (input/output), 28–30
INT signal, 258
integer arithmetic
expr command, 119–120
integer types, 304–305
numbers in different bases, 305–306
overview, 303–304
integer operators, 140–142
integer types, 304–305
interactive shell features
aliases
defining, 307–309
removing, 309
arrays, 309–314
cd command, 317–318
command history, accessing
emacs line edit mode, 296–298
fc command, 301
history command, 300–301
r command, 301–303
vi line edit mode, 294–296
command-line editing
command history, 292
emacs line edit mode, 296–300
overview, 291
vi line edit mode, 292–296
ENV file, 290–291
functions
automatically loaded functions, 303
local variables, 303
integer arithmetic
integer types, 304–305
numbers in different bases, 305–306
overview, 303–304
job control, 315–317
order of search, 319
shell, specifying, 290
tilde substitution, 318–319
internal field separators, 251–254
interpreted programming language, 50
I/O (input/output) redirection
<&- construct, 262
>&- construct, 262
input redirection
< (left arrow), 331
exec command, 230–231, 262
in-line input redirection
shell archive creation, 264–267
syntax, 262–264
POSIX shell, 331
standard I/O (input/output), 28–30
loops, 177–178
output redirection
exec command, 230–231, 262
overview, 30–32
standard output, closing, 262
overview, 48–49, 331–332
in programs, 94
standard error, writing to, 261–262
standard I/O (input/output), 28–30

ison program, 122

J

jobs
asynchronous jobs, 257
bringing to foreground, 342
job control, 315–317
job numbers, 37
killing, 347
printing list of, 347
referencing, 333–334
sending to background, 316–317
stopped jobs, 316–317
stopping, 334
waiting for, 358
waiting for completion
$! variable, 257–258
wait command, 257

jobs command, 315, 347

K

kernel, 1, 39
keyword parameters. See variables (shell)
kill command, 315, 347
killing jobs, 347

Korn, David, 289, 360
Korn shell. See also nonstandard shell features
compatibility summary, 319–320
history of, 289
Web documentation, 360
kornshell.com website, 360

L

-L file operator, 142–143
-le operator, 140–142
left-shifting positional parameters, 128–129

line edit modes
emacs
command history, accessing, 296–298
overview, 296–298
overview, 291
vi
command history, accessing, 294–296
overview, 292–294

in-line input redirection
shell archive creation, 264–267
syntax, 262–264

LINENO variable, 324

lines
cutting, 64–66
deleting, 73
duplicate lines, eliminating
sort command, 84
uniq command, 88–89
line continuation, 112
pasting
from different files, 68–69
output delimiters, 69–70
from same file, 70
pattern matching
beginning of line, 53
end of line, 53–54
sorting
arbitrarily, 66
delimiter characters, 67
duplicate lines, eliminating, 84
to output file, 85
overview, 84
reverse order, 85
skipped fields, 87

linking files, 20–23

Linux resources
books, 360–361
online documentation, 359
overview, 359
Web documentation, 360

listall program, 283–284

listing
files, 7, 15–17
phonebook file entries, 283–284
variables, 247

In command, 20–23
local variables, 209–210, 303
logical AND operator (-a), 143–144
logical negation operator (!), 143, 322
logical OR operator (-o), 144
login cycle, 44
login shell, 40–43
looking up phonebook entries, 124–125, 278
for loops, 342–343

loops
body of, 164
breaking out of, 174–176
executing in background, 177
for
$* variable, 166
$@ variable, 166–167
overview, 163–166, 342–343
without in element, 167–168
getopts command, 180–184
I/O redirection, 177–178
piping data into and out of, 178–179

skipping remaining commands in,
176–177
terminating, 336
typing on one line, 179–180
until, 170–174
while, 168–170, 358

ls command, 7, 15–17
-lt operator, 140–142
lu program, 124–125, 278

M

MAIL variable, 324
MAILCHECK variable, 324
MAILPATH variable, 324
man command, 359

matched characters, saving, 61–63

matching patterns
any character, 51–53
beginning of line, 53
case command, 155–156, 336–337
character sets, 55–57
duplicate entries, 89–90
end of line, 53–54
filename substitution, 25–27
grep command
-l option, 82–83
-n option, 83
overview, 78–81
regular expressions, 81–82
-v option, 82

matched characters, saving, 61–63
overview, 51
parameter substitution, 242–244
precise number of subpatterns, 59–61
summary of regular expression characters, 63–64
zero or more characters, 57–59
nonstandard shell features

mathematical equation solver (expr), 119–120

menu-driven phone program (rolo)

$n$ variable, 198–199
add program, 277
change program, 281–283
data formatting, 273–274
display program, 278–279
final code listing, 274–277
initial code listing, 193–194
listall program, 283–284
lu program, 278
rem program, 280–281
revised code listing, 196–198
sample output, 284–287
sample runs, 195–196
temporary files, 198–199
messages (error), displaying, 245
minus sign (−). See hyphen (−)
mkdir command, 17–18
monitor option (set command), 331–332
monitor shell mode, 351
moving files between directories, 19
multiple commands on same line, 36
mv command, 8–9, 19
mybasename program, 244
mycp program

  echo escape characters, 187–188
  final code listing, 190–193
  initial code listing, 185–187
  revised code listing, 188–190

myecho program

-n string operator, 137, 138

names

  filename substitution
    asterisk (*), 24–25
  overview, 27, 47, 331
  POSIX shell, 331
  question mark (?), 25–27
  variables, 101–103

filenames

  allowed characters, 6
  changing, 8–9
  spaces in, 27
  special characters, 28

pathnames, 10–12

-ne operator, 140–142

nesting if statements, 148–149

newgrp command, 347–348
newline character, 45
noclobber shell mode, 351
noexec shell mode, 351
noglob shell mode, 351
nolog shell mode, 351

nonstandard shell features

  aliases
    defining, 307–309
    removing, 309
  arrays, 309–314
  command history, accessing
    emacs line edit mode, 296–298
    fc command, 301
    history command, 300–301
    r command, 301–303
    vi line edit mode, 294–296
  command-line editing
    command history, 292
    emacs line edit mode, 296–300
  overview, 291
    vi line edit mode, 292–296
  ENV file, 290–291
  functions
automatically loaded functions, 303
local variables, 303
integer arithmetic
integer types, 304–305
numbers in different bases, 305–306
overview, 303–304
job control, 315–317
numbers, 304–305
order of search, 319
shell, specifying, 290
tilde substitution, 318–319
non-zero exit status, 131
nounset shell mode, 351
null command (.), 160–161
null values, 100–101
number program, 154–155, 201–202
number 2 program, 253
numbers
in different bases, 305–306
job numbers, 37
signal numbers, 258, 355

integer operators, 140–142
logical AND operator (-a), 143–144
logical negation operator (!), 143
logical OR operator (-o), 144
string operators, 135–139
test operators, 353–354
options (command), 8
ordinary files, 6
O'Reilly & Associates, 360–361
output
output redirection
> (right arrow), 30–32
exec command, 230–231, 262
POSIX shell, 331–332
standard output, closing, 262
standard I/O (input/output), 28–30
output delimiters
paste command, 69–70
sort command, 88

packages, Cygwin, 360
parameters. See also variables (shell)
overview, 239
parameter substitution
${parameter}, 239–240
${parameter:+value}, 242
${parameter:=-value}, 241
${parameter:?value}, 241–242
overview, 324–325
pattern matching, 242–244
positional parameters
definition of, 239
left-shifting, 128–129
overview, 322
reassigning values to, 239, 247–248
setting, 350
shifting left, 352
substitution, 121–122
special parameters, 323–324

parent processes, 257

parentheses in test command, 144

parsing phase, 44

passing
arguments
$# variable, 122–123
$* variable, 123–124
${[n]} special variable, 128
phonebook file example, 124–128
positional parameters, 121–122
shift command, 128–129
variables to subshells, 234–235

passwords, entering at login, 40

paste command
-d option, 69–70
overview, 68–69
-s option, 70

pasting lines
from different files, 68–69
output delimiters, 69–70
from same file, 70

PATH variable, 217–224, 324

pathnames, 10–12

pattern matching
any character, 51–53
beginning of line, 53
case command, 155–156, 336–337
duplicate entries, 89–90
dot (.) command, 227–230, 334–335
end of line, 53–54
filename substitution, 25–27
grep command

pipe symbol (|)

Pearson books, 361

percent sign (%)
job control, 315
pattern matching, 242–243
with printf command, 202

period (.)
dot (.) command, 227–230, 334–335
pattern matching, 51–53, 63

perl command, 91

phonebook file. See also rolo (Rolodex) program
adding entries to, 125–127, 277
displaying entries from, 278–279
editing entries in, 281–283
listing entries in, 283–284
looking up entries in, 124–125, 278
removing entries from, 127–128, 280–281

PIDs (process IDs), 37, 199

pipe symbol (|)
plus sign (+)
  job control, 315
  pattern matching, 63
  printf format specification modifier, 206
pointers to variables, 257
positional parameters
  definition of, 239
  left-shifting, 128–129
  overview, 322
  reassigning values to, 239, 247–248
  setting, 350
  shifting left, 352
  substitution, 121–122
POSIX shell
  compatibility summary, 319–320
  overview, 1
  startup, 321
  subshell execution, 332
  vi line edit mode, 326–329
  Web documentation, 360
pound sign (#)
  comments, 96
  pattern matching, 243
  printf format specification modifier, 206
PPID variable, 324
prargs program, 169–170
precedence of operators, 330
precise number of subpatterns, matching, 59–61
precision modifier (printf), 205–206
printf command
  example, 206–207
  format specification characters, 202–205
  format specification modifiers, 205–206
  syntax, 202
printing
  command information, 356
  date/time, 5, 95–96
  formatted output
    example, 206–207
    format specification characters, 202–205
    format specification modifiers, 205–206
    printf command syntax, 202
  list of active jobs, 347
  to working directory, 348
process IDs (PIDs), 37, 199
processes
  definition of, 43
  parent/child, 257
  returning information about, 37
  waiting for completion
    $! variable, 257–258
    wait command, 257
.profile file, 235–236
  on program, 132–135, 145–147, 170–171
programs
  add, 125–127, 277
  addi, 200
  args, 122–123
  arguments, passing
    $# variable, 122–123
    $* variable, 123–124
    ${[a]} special variable, 128
  phonebook file example, 124–128
  positional parameters, 121–122
  shift command, 128–129
cdtest, 225
change, 281–283
command files, 93–96
  comments, 96
cctype, 155–156, 158–159
db, 227–229
debugging, 157–159
display, 278–279
execution, 45–47
getty. See shells
greetings, 149–151, 159–160
init, 40–43
ison, 122
listall, 283–284
lu, 124–125, 278
mybasename, 244
mycp
  echo escape characters, 187–188
  final code listing, 190–193
  initial code listing, 185–187
  revised code listing, 188–190
number, 154–155
number2, 253
on, 132–135, 145–147, 170–171
prargs, 169–170
rem, 147–148, 151–153, 280–281
reverse, 311
rolo (Rolodex)
  $$ variable, 198–199
  add program, 277
  change program, 281–283
  data formatting, 273–274
  display program, 278–279
  final code listing, 274–277
  fun, 270–271
  initial code listing, 193–194
  listall program, 283–284
  lu program, 278
  PATH variable, 221–224
  rem program, 280–281
  revised code listing, 196–198
  rolosubs file, 264–266
  sample output, 284–287
  sample runs, 195–196
  temporary files, 198–199
  run, 95, 121, 164
  shar, 267
shell variables
  arithmetic expansion, 103–104
  assigning values to, 97, 322, 333
  definition of, 97
  displaying values of, 98–100
  exported variables, 332, 340–341
  filename substitution, 101–103
  HISTSIZE, 326
  null values, 100–101
  readonly variables, 349
  table of, 323–324
  undefined variables, 100–101
stats, 95–96
trace mode, turning on/off, 246
twhile, 169
vartest, 209
vartest2, 210
vartest3, 212
vartest4, 213–214
waitfor, 171–174, 180–184
words, 249–250
ps command, 37
PS1 variable, 216, 324
PS2 variable, 216, 324
PS4 variable, 324
pseudo-terminals, 40
pseudo-tty, 40
pwd command, 12, 95–96, 348
PWD variable, 324
question mark (?)
  filename substitution, 25–27, 47, 331
  pattern matching, 242, 336
quote characters
  back quote (‘), 114–115
  backslash (\)
    escaping characters with, 111–114
    inside double quotes, 112–114
  line continuation, 112
  overview, 111–112
  double quotes (“), 109–111
  overview, 329
  single quotes (‘), 105–108
  smart quotes, 119

r command, 301–303
-ﬁle operator, 142–143
race conditions, 199
read command
  exit status, 199–202
  menu-driven phone program (rolo)
    $ variable, 198–199
    initial code listing, 193–194
    revised code listing, 196–198
    sample runs, 195–196
    temporary files, 198–199
  mycp program
    echo escape characters, 187–188
    final code listing, 190–193
    initial code listing, 185–187
    revised code listing, 188–190
    read command syntax, 185
readonly command, 254, 349
read-only variables, 254, 349
reassigning values to positional parameters, 239, 247–248
redirection (I/O)
  <&- construct, 262
  >&- construct, 262
  input redirection
    < (left arrow), 331
    exec command, 230–231, 262
    in-line input redirection, 262–267
    overview, 331
    POSIX shell, 331
    shell archive creation, 264–267
    standard I/O (input/output), 28–30
    loops, 177–178
  output redirection
    exec command, 230–231, 262
    overview, 30–32
    standard output, closing, 262
    overview, 48–49, 331–332
    POSIX shell, 331–332
    in programs, 94

reading data
  exit status, 199–202
  menu-driven phone program (rolo)
    $ variable, 198–199
    initial code listing, 193–194
    revised code listing, 196–198
    sample runs, 195–196
    temporary files, 198–199
  mycp program
    echo escape characters, 187–188
    final code listing, 190–193
    initial code listing, 185–187
    revised code listing, 188–190
  read command syntax, 185
running rolo program

standard error, writing to, 261–262
standard I/O (input/output), 28–30
redirection append (>>) characters, 31–32
re-entry of commands, 326
references
  books, 360–361
  online documentation, 359
  overview, 359
  Web documentation, 360
referencing jobs, 333–334
regex. See regular expressions
registering your book, 3
regular expressions
  [ ] construct, 55–57
  \(\) construct, 61–63
  \{\} construct, 59–61
  asterisk (*), 57–59
  caret (^), 53
  dollar sign ($), 53–54
  grep command, 81–82
  overview, 51
  period (.), 51–53
  summary table, 61–63
rem program, 147–148, 151–153, 280–281
removing
  aliases, 309
  characters from input stream, 77–78
directories, 22–23
duplicate lines
  sort command, 84
uniq command, 88–89
files, 9
function definitions, 271
lines, 73
phonebook file entries, 127–128, 280–281
renaming files, 8–9
resetting traps, 261
resources
  books, 360–361
  online documentation, 359
  overview, 359
  Web documentation, 360
return command, 271, 349
reverse program, 311
reversing sort order, 85
Ritchie, Dennis, 1
rm command, 9
rmdir command, 22–23
rolo (Rolodex) program
  $$ variable, 198–199
  add program, 277
  change program, 281–283
data formatting, 273–274
display program, 278–279
final code listing, 274–277
functions, 270–271
initial code listing, 193–194
doctest program, 283–284
lu program, 278
PATH variable, 221–224
rem program, 280–281
revised code listing, 196–198
rolsubs file, 264–266
sample output, 284–287
temporary files, 198–199
Rolodex program. See rolo (Rolodex) program
rolsubs file, 264–266
run program, 95, 121, 164
running rolo program, 195–196
shell variables. See also parameters
arithmetic expansion, 103–104
assigning values to, 97, 322, 333
definition of, 97
displaying values of, 98–100
ENV, 290–291
exported variables, 211–216, 332, 340–341
filename substitution and,
101–103
finding number of characters stored in,
244
HISTSIZE, 326
HOME, 217
IFS, 251–254
listing, 247
local variables, 209–210, 303
null values, 100–101
passing to subshells,
234–235
PATH, 217–224
pointers, creating, 257
PS1, 216
PS2, 216
read-only variables, 254, 349
special variables
$? variable, 132–135
$! variable, 257–258
#$ variable, 122–123
$* variable, 123–124, 166
$@ variable, 166–167
$[n] variable, 128
$0 variable, 245
table of, 323–324
TERM, 236–237
TZ, 236–237
undefined variables, 100–101
unsetting, 254

S
-s file operator, 142–143
%s format specification character,
203
saving matched characters, 61–63
scanning command line twice before
executing, 255–257
search order, 319
searching. See pattern matching
searching files with grep
-l option, 82–83
-n option, 83
overview, 78–81
regular expressions, 81–82
-v option, 82
sed command
d command, 73
examples, 73
-n option, 72
overview, 70–72
semicolon (;), 36, 179–180, 321
sending commands to background,
36–37
sequences of characters
double quotes ("), 109–111
single quotes (‘), 105–108
set command
-- option, 248–250
IFS variable, 251–254
monitor option, 331
with no arguments, 247
overview, 239, 321, 350–351
positional parameters, reassigning,
247–248
-x option, 246
shar program, 267
shell archives, creating, 264–267
shells
Bash. See also nonstandard shell features
compatibility summary, 319–320
history of, 289
Web documentation, 360
compatibility summary, 319–320
current shell, executing files in, 227–230
definition of, 1
Korn shell. See also nonstandard shell features
compatibility summary, 319–320
history of, 289
Web documentation, 360
login shell, 40–43
POSIX shell. See POSIX shell responsibilities of
environment control, 49
filename substitution, 47
interpreted programming language, 50
I/O redirection, 48–49
overview, 44–45
pipelines, 49
program execution, 45–47
variable substitution, 47
specifying, 289–290
subshells
definition of, 43
environment, 210–211
overview, 227
terminating, 340
typing commands to, 43–44
shift command, 128–129, 352
signals
handling with trap command
execution with no arguments, 259–260
ignore signals, 260
overview, 258–259
signal numbers, 258
trap reset, 261
ignoring, 260
numbers, 355
single quotes (‘), 105–108
skipping commands in loops, 176–177
fields during sort, 87
smart quotes, 119
sort command
-k2n option, 87
-n option, 86
-o option, 85
other options, 88
overview, 84
-r option, 85
-t option, 87–88
-u option, 84
sorting lines
arithmetic, 86
delimiter characters, 87–88
duplicate lines, eliminating, 84
to output file, 85
overview, 84
reverse order, 85
skipped fields, 87
spaces in filenames, 27
sparse arrays, 310
special files, 6
special variables
$? variable, 132–135
$! variable, 257–258
$# variable, 122–123
$* variable, 123–124, 166
$@ variable, 166–167
special variables

$[n]$ variable, 128
$0 variable, 245
table of, 323–324
specifying shell, 289–290
standard error
overview, 35
writing to, 261–262
standard I/O (input/output)
closing, 262
deleting from input stream, 77–78
overview, 28–30
redirecting, 230–231
redirection, 261–262
translating characters from, 74–77
standard shell. See POSIX shell
starting up POSIX shell, 321
statements
&& construct, 161–162
|| construct, 161–162
comments, 96
if
elif construct, 148–151
derlf construct, 145–147
exit command, 147–148
exit status, 131–135
nesting, 148–149
syntax, 131
testing conditions in, 131–144
testing conditions in
alternative format for test, 139–140
file operators, 142–143
integer operators, 140–142
logical AND operator (-a), 143–144
logical negation operator (!), 143
logical OR operator (-o), 144
overview, 135
string operators, 135–139
stats program, 95–96
status, exit
$? variable, 132–135
definition of, 321
non-zero values, 131
overview, 131–132
zero values, 131
stopped jobs, 316–317, 334
storing values in variables, 97
stream editor (sed)
command overview, 70–72
d command, 73
examples, 73
-n option, 72
string operators, 135–139
subpatterns, matching, 59–61
subscripts, 309
subshells
(.) construct, 231–234
[ .; ] construct, 231–234
definition of, 43
dot (.) command, 227–230
evironment, 210–211
exec command, 227–230
execution, 332
overview, 227
passing variables to, 234–235
substitution
command substitution
$.() construct, 115–118
back quote (‘), 114–115
definition of, 112
expr command, 119–120
filename substitution
asterisk (*), 24–25
overview, 47
POSIX shell, 331
question mark (?), 25–27
shell variables, 101–103

parameter substitution
$\{parameter\}, 239–240
$\{parameter:+value\}, 242
$\{parameter=value\}, 241
$\{parameter=value\}, 240
$\{parameter?value\}, 241–242
overview, 324–325
pattern matching, 242–244
positional parameters, 121–122
	tilde substitution, 318–319, 329
variable substitution, 47, 98–100

suspending jobs, 316

symbolic links, 21–23

temporary files, 198–199

TERM, 236–237, 258

terminal, 28

terminating
functions, 271
jobs, 315
loops, 336
shell program, 340

test command
alternative format, 139–140
file operators, 142–143
integer operators, 140–142
logical AND operator (-a), 143–144
logical negation operator (!), 143
logical OR operator (-o), 144
overview, 135
parentheses, 144
string operators, 135–139
test command syntax, 135

text
ASCII characters, octal values of, 75
character sequences
double quotes ("), 109–111
single quotes (’), 105–108
cutting, 64–66
deleting
with sed, 73
with tr command, 77–78
echoing, 6
escaping, 111–112
filenames
allowed characters, 6
special characters, 28
line continuation, 112
pasting, 68–70
pattern matching
any character, 51–53
beginning of line, 53
character sets, 55–57
end of line, 53–54
filename substitution, 25–27
matched characters, saving, 61–63
parameter substitution, 242–244
precise number of characters, 59–61
zero or more characters, 57–59
sorting, 84–88
translating from standard input, 74–77
Thompson, Ken, 1
tilde substitution, 318–319, 329
time
  printing, 5
time zone, determining, 237
times command, 354, 355–356
tools. See also commands
tr command
  -d option, 77–78
  examples, 78
  octal values of ASCII characters, 75
  overview, 74–76
  -s option, 76–77
trace mode, turning on/off, 246
  translating characters from standard input, 74–77
trap command
  execution with no arguments, 259–260
  ignored signals, 260
  overview, 258–259
  signal numbers, 258
  trap reset, 261
Trojan horse, 218–219
true command, 356
turning on/off trace mode, 246
twhile program, 169
type command, 271
types, integer, 304–305
typing loops on one line, 179–180
TZ variable, 236–237

U
%u format specification character, 203
umask command, 356
unalias command, 309, 356
unary file operators, 142–143
unary logical negation operator (!), 143
undefined variables, 100–101
underscore (_), 322
uniq command
  -c option, 90
  -d option, 89–90
  overview, 88–89
Unix
  development of, 1
  resources
    books, 360–361
    online documentation, 359
    overview, 359
    Web documentation, 360
  strengths of, 1
unix.org website, 360
unset command, 254, 271, 357
until command, 170–174, 357
users, returning information about, 5–6
utilities, 39. See also commands

V
values
  assigning to variables, 97
  reassigning to positional parameters, 239, 247–248
variables (shell). See also parameters
  arithmetic expansion, 103–104
  assigning values to, 97, 322, 333
  definition of, 97
  displaying values of, 98–100
ENV, 290–291
exported variables, 211–216, 332, 340–341
filename substitution, 101–103
finding number of characters stored in, 244
HISTSIZE, 326
HOME, 217
IFS, 251–254
listing, 247
local variables, 209–210, 303
null values, 100–101
passing to subshells, 234–235
PATH, 217–224
pointers, creating, 257
PS1, 216
PS2, 216
read-only variables, 254
read-only variables, 349
special variables
$? variable, 132–135
$! variable, 257–258
$# variable, 122–123
$* variable, 123–124, 166
$@ variable, 166–167
${[n]} variable, 128
$0 variable, 245
substitution, 47, 98–100
table of, 323–324
TERM, 236–237
TZ, 236–237
undefined variables, 100–101
unsetting, 254
vartest program, 209
vartest2 program, 210
vartest3 program, 212
vartest4 program, 213–214
verbose shell mode, 351
vi line edit mode
command history, accessing, 294–296
overview, 292–294, 326–329, 351
-w file operator, 142–143
wait command, 257, 358
waiting for job completion
$! variable, 257–258
overview, 358
wait command, 257
wc command, 7, 95–96
Web documentation, 360
Web Edition of book, 3
websites
Cygwin, 360
cygwin.com, 360
Free Software Foundation, 360
Korn shell, 360
The Open Group, 360
while loops, 168–170, 358
whitespace, 45, 321
who command, 5–6
width modifier (printf), 205–206
words, counting, 7
words program, 249–250
working directory
definition of, 10
displaying, 12
printing to, 348
writing to standard error, 261–262
<table>
<thead>
<tr>
<th>X</th>
<th>Y-Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x file operator, 142-143</td>
<td>-z string operator, 137, 138</td>
</tr>
<tr>
<td>%X format specification character, 203</td>
<td>zero exit status, 131</td>
</tr>
<tr>
<td>%x format specification character, 203</td>
<td>zero or more characters, matching, 57-59</td>
</tr>
<tr>
<td>-x option, debugging with, 157-159</td>
<td></td>
</tr>
<tr>
<td>xtrace mode, 351</td>
<td></td>
</tr>
</tbody>
</table>