

THE EXPERT'S VOICE® IN OPEN SOURCE

SECOND EDITION

Shell Scripting Recipes

A Problem-Solution Approach

*POSIX-COMPLIANT SCRIPTS FOR BASH,
KSH, SH, AND MORE*

Chris F. A. Johnson, Jayant Varma

Apress®

Shell Scripting Recipes

A Problem-Solution Approach

Second Edition



Chris F. A. Johnson

Jayant Varma

Apress®

Shell Scripting Recipes, Second Edition

Copyright © 2015 by Chris F. A. Johnson and Jayant Varma

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0221-0

ISBN-13 (electronic): 978-1-4842-0220-3

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Louise Corrigan

Editorial Board: Steve Anglin, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson,

Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper,

Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade,

Steve Weiss

Coordinating Editor: Mark Powers

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781484202210. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

*This book is dedicated to my parents,
who would have been quite proud to see this book.*

—Jayant Varma

Contents at a Glance

About the Authors.....	xxi
Acknowledgments.....	xxiii
■ Chapter 1: The POSIX Shell and Command-Line Utilities.....	1
■ Chapter 2: Playing with Files: Viewing, Manipulating, and Editing Text Files	43
■ Chapter 3: String Briefs.....	63
■ Chapter 4: What's in a Word?	83
■ Chapter 5: Scripting by Numbers	107
■ Chapter 6: Loose Names Sink Scripts: Bringing Sanity to Filenames	139
■ Chapter 7: Treading a Righteous PATH	159
■ Chapter 8: The Dating Game	167
■ Chapter 9: Good Housekeeping: Monitoring and Tidying Up File Systems	187
■ Chapter 10: Screenplay: The screen-funcs Library	201
■ Chapter 11: Aging, Archiving, and Deleting Files	229
■ Chapter 12: Covering All Your Databases	239
■ Chapter 13: Home on the Web	259
■ Chapter 14: Taking Care of Business.....	281
■ Chapter 15: Random Acts of Scripting	297
■ Chapter 16: A Smorgasbord of Scripts	317

■ **Chapter 17: Script Development Management..... 333**

■ **Appendix A: Internet Scripting Resources..... 343**

Index..... 347

Contents

About the Authors.....	xxi
Acknowledgments.....	xxiii
■ Chapter 1: The POSIX Shell and Command-Line Utilities.....	1
Shell Commands	1
echo.....	1
printf.....	2
set.....	3
shift.....	5
type.....	5
getopts.....	6
case	6
eval	7
local	7
Parameters and Variables	7
Positional Parameters.....	7
Special Parameters	7
standard-vars—A Collection of Useful Variables.....	9
Patterns.....	10
Pathname Expansion	10
Regular Expressions	11
Parameter Expansion	12
The Bourne Shell Expansions	12
POSIX Parameter Expansions	14
Shell-Specific Expansions, bash2, and ksh93	16

Shell Arithmetic	16
Aliases	17
Sourcing a File	17
Functions	17
Functions Are Fast	18
Command Substitution Is Slow	18
Using the Functions in This Book	19
standard-funcs: A Collection of Useful Commands	19
1.1 get_key—Get a Single Keystroke from the User	19
1.2 getline—Prompt User to Enter a Line	21
1.3 press_any_key—Prompt for a Single Keypress	22
1.4 menu1—Print a Menu and Execute a Selected Command	22
1.5 arg—Prompt for Required Argument If None Supplied	24
1.6 die—Print Error Message and Exit with Error Status	25
1.7 show_date—Display Date in D[D] MMM YYYY Format	25
1.8 date_vars—Set Date and Time Variables	27
1.9 is_num—Is This a Positive Integer?	28
1.10 abbrev_num—Abbreviate Large Numbers	30
1.11 commas—Add Thousands Separators to a Number	31
1.12 pr1—Print Arguments, One to a Line	32
1.13 checkdirs—Check for Directories; Create If Necessary	33
1.14 checkfiles—Check That a Directory Contains Certain Files	34
1.15 zpad—Pad a Number with Leading Zeroes	35
1.16 cleanup—Remove Temporary Files and Reset Terminal on Exit	36
The Unix Utilities	37
cat: Concatenate Files to the Standard Output	37
sed: A Text Stream Editor	37
awk: Pattern Scanning and Processing Language	38
grep: Print Lines Matching a Regular Expression	38
date: Show or Set the System Date	39
tr: A Character Translation Utility	39

wc: Count Characters, Words, and Lines in a File.....	39
file: Determine the File Type	40
ls: Sort and Provide Details About Files.....	40
uniq: Remove Consecutive Duplicate Lines.....	40
sudo: Execute Commands as the Superuser	41
split: Divide a File into Equal-Sized Pieces.....	41
which: Show the Full Path to a Command.....	41
gs, gv: Render, Convert, or View PostScript and PDF Files	41
Summary	41
■ Chapter 2: Playing with Files: Viewing, Manipulating, and Editing Text Files	43
End of the Line for OS Differences: Converting Text Files	43
2.1 dos2unix—Convert Windows Text Files to Unix	44
2.2 unix2dos—Convert a Unix File to Windows.....	45
2.3 mac2unix—Convert Macintosh Files to Unix	46
2.4 unix2mac—Convert Unix Files to Mac Format.....	46
2.5 dos2mac—Convert Windows Files to Macintosh	47
2.6 mac2dos—Convert Macintosh Files to Windows	47
Displaying Files	48
2.7 prn—Print File with Line Numbers.....	48
2.8 prw—Print One Word per Line	51
2.9 wbl—Sort Words by Length	53
Formatting File Facts	54
2.10 finfo—File Information Display	54
2.11 wfreq—Word Frequency	58
2.12 lfreq—Letter Frequency	58
2.13 fed—A Simple Batch File Editor.....	60
Summary	62

■ **Chapter 3: String Briefs** **63**

Character Actions: The char-funcs Library **63**

 3.1 chr—Convert a Decimal Number to an ASCII Character..... **63**

 3.2 asc—Convert a Character to Its Decimal Equivalent..... **65**

 3.3 nxt—Get the Next Character in ASCII Sequence **66**

 3.4 upr—Convert Character(s) to Uppercase **67**

 3.5 lwr—Convert Character(s) to Lowercase **69**

String Cleaning: The string-funcs Library..... **71**

 3.6 sub—Replace First Occurrence of a Pattern..... **71**

 3.7 gsub—Globally Replace a Pattern in a String **73**

 3.8 repeat—Build a String of a Specified Length..... **75**

 3.9 index, rindex—Find Position of One String Within Another **76**

 3.10 substr—Extract a Portion of a String **78**

 3.11 insert_str—Place One String Inside Another **80**

Summary **82**

■ **Chapter 4: What’s in a Word?** **83**

Finding and Massaging Word Lists..... **84**

wf-funcs: WordFinder Function Library **84**

 4.1 write_config—Write User’s Information to the Configuration File **84**

 4.2 do_config—Check For and Source Default Configuration File..... **85**

 4.3 set_sysdict—Select the Dictionary Directory **86**

 4.4 mkwsig—Sort Letters in a Word **87**

 4.5 wf-clean—Remove Carriage Returns and Accents **88**

 4.6 wf-compounds—Squish Compound Words and Save with Lengths **89**

 4.7 wf-setup—Prepare Word and Anagram Lists..... **92**

Playing with Matches **93**

 4.8 wf—Find Words That Match a Pattern **94**

 4.9 wfb—Find Words That Begin with a Given Pattern **95**

 4.10 wfe—Find Words That End with a Given Pattern **96**

 4.11 wfc—Find Words That Contain a Given Pattern **98**

 4.12 wfit—Find Words That Fit Together in a Grid..... **99**

4.13 anagram—Find Words That Are Anagrams of a Given Word	102
4.14 aplus—Find Anagrams of a Word with a Letter Added.....	103
4.15 aminus—Remove Each Letter in Turn and Anagram What's Left.....	104
Summary	106
■ Chapter 5: Scripting by Numbers	107
The math-funcs Library.....	107
5.1 fpmul—Multiply Decimal Fractions.....	108
5.2 int—Return the Integer Portion of a Decimal Fraction	112
5.3 round—Round the Argument to the Nearest Integer.....	112
5.4 pow—Raise a Number to Any Given Power	113
5.5 square—Raise a Number to the Second Power	115
5.6 cube—Raise a Number to the Third Power.....	115
5.7 calc—A Simple Command-Line Calculator	116
Adding and Averaging	117
5.8 total—Add a List of Numbers.....	117
5.9 mean—Find the Arithmetic Mean of a List of Numbers	118
5.10 median—Find the Median of a List of Numbers	120
5.11 mode—Find the Number That Appears Most in a List	121
5.12 range—Find the Range of a Set of Numbers	123
5.13 stdev—Finding the Standard Deviation	124
Converting Between Unit Systems	126
5.14 conversion-funcs—Converting Metric Units	126
5.15 conversion—A Menu System for Metric Conversion.....	133
Summary	137
■ Chapter 6: Loose Names Sink Scripts: Bringing Sanity to Filenames	139
What's in a Name?	139
POSIX Portable Filenames	140
OK Filenames.....	140
Functioning Filenames: The filename-funcs Library	141
6.1 basename—Extract the Last Element of a Pathname.....	141

6.2	dirname—Return All but the Last Element of a Pathname	143
6.3	is_pfname—Check for POSIX Portable Filename.....	144
6.4	is_OKfname—Check Whether a Filename Is Acceptable	145
6.5	pfname—Convert Nonportable Characters in Filename.....	146
6.6	OKfname—Make a Filename Acceptable	147
6.7	is_whitespace—Does the Filename Contain Whitespace Characters?.....	148
6.8	whitespace—Fix Filenames Containing Whitespace Characters	149
6.9	is_dir—Is This a Directory I See Before Me?	150
6.10	nodoublechar—Remove Duplicate Characters from a String	151
6.11	new_filename—Change Filename to Desired Character Set.....	152
6.12	fix_pwd—Fix All the Filenames in the Current Directory	153
6.13	fixfname—Convert Filenames to Sensible Names.....	154
	Summary	157
■	Chapter 7: Treading a Righteous PATH	159
	The path-funcs Library	159
7.1	path—Display a User's PATH.....	159
7.2	unslash—Remove Extraneous Slashes.....	160
7.3	checkpath—Clean Up the PATH Variable.....	161
7.4	addpath—Add Directories to the PATH Variable.....	163
7.5	rmpath—Remove One or More Directories from \$PATH	165
	Summary	166
■	Chapter 8: The Dating Game	167
	The date-funcs Library	168
8.1	split_date—Divide a Date into Day, Month, and Year	168
8.2	is_leap_year—Is There an Extra Day This Year?.....	169
8.3	days_in_month—How Many Days Hath September?	170
8.4	Julian Dates	172
8.5	dateshift—Add or Subtract a Number of Days.....	174
8.6	diffdate—Find the Number of Days Between Two Dates	175

8.7 day_of_week—Find the Day of the Week for Any Date	176
8.8 display_date—Show a Date in Text Format	178
8.9 parse_date—Decipher Various Forms of Date String	180
8.10 valid_date—Where Was I on November 31st?	185
Summary	186
■ Chapter 9: Good Housekeeping: Monitoring and Tidying Up File Systems	187
9.1 dfcmp—Notify User of Major Changes in Disk Usage.....	187
How It Works.....	187
9.2 symfix—Remove Broken Symbolic Links	191
How It Works.....	191
9.3 sym2file—Converts Symbolic Links to Regular Files	193
How It Works.....	193
9.4 zrm—Remove Empty Files.....	195
How It Works.....	195
9.5 undup—Remove Duplicate Files.....	196
How It Works.....	196
9.6 lsr—List the Most Recent (or Oldest) Files in a Directory	198
How It Works.....	198
Summary	200
■ Chapter 10: Screenplay: The screen-funcs Library	201
10.1 screen-vars—Variables for Screen Manipulation	202
How It Works.....	202
10.2 set_attr—Set Screen-Printing Attributes.....	204
How It Works.....	204
10.3 set_fg, set_bg, set_fgbg—Set Colors for Printing to the Screen.....	205
How It Works.....	205
10.4 cls—Clear the Screen	206
How It Works.....	206
Notes	206

10.5 printat—Position Cursor by Row and Column.....	207
How It Works.....	207
Notes	209
10.6 put_block_at—Print Lines in a Column Anywhere on the Screen	211
How It Works.....	211
10.7 get_size—Set LINES and COLUMNS Variables	213
How It Works.....	213
10.8 max_length—Find the Length of the Longest Argument.....	213
How It Works.....	214
10.9 print_block_at—Print a Block of Lines Anywhere on the Screen	214
How It Works.....	214
10.10 vbar, hbar—Print a Vertical or Horizontal Bar	216
How It Works.....	216
10.11 center—Center a String on N Columns.....	217
How It Works.....	217
10.12 flush_right—Align String with the Right Margin.....	218
How It Works.....	218
10.13 ruler—Draw a Ruler Across the Width and Height of the Window	219
How It Works.....	219
10.14 box_block, box_block_at—Print Text Surrounded by a Box	221
How It Works.....	221
10.15 clear_area, clear_area_at—Clear an Area of the Screen	223
How It Works.....	223
10.16 box_area, box_area_at—Draw a Box Around an Area	224
How It Works.....	225
10.17 screen-demo—Saving and Redisplaying Areas of the Screen	226
How It Works.....	226
Summary.....	228

■ Chapter 11: Aging, Archiving, and Deleting Files	229
11.1 date-file—Add a Datestamp to a Filename.....	229
How It Works.....	229
Notes	233
11.2 rmold—Remove Old Files	233
How It Works.....	233
11.3 keepnewest—Remove All but the Newest or Oldest Files.....	236
How It Works.....	236
Summary.....	238
■ Chapter 12: Covering All Your Databases	239
12.1 lookup—Find the Corresponding Value for a Key	240
How It Works.....	241
shdb-funcs: Shell Database Function Library	242
12.2 load_db—Import Database into Shell Array.....	243
How It Works.....	243
Notes	243
12.3 split_record—Split a Record into Fields	243
How It Works.....	244
12.4 csv_split—Extract Fields from CSV Records	245
How It Works.....	245
12.5 put_record—Assemble a Database Record from an Array	247
How It Works.....	247
12.6 put_csv—Assemble Fields into a CSV Record.....	248
How It Works.....	248
12.7 db-demo—View and Edit a Password File.....	249
How It Works.....	249
PhoneBase: A Simple Phone Number Database.....	253
How It Works.....	253

12.8 ph—Look Up a Phone Number.....	254
12.9 phadd—Add an Entry to PhoneBase.....	255
12.10 phdel—Delete an Entry from PhoneBase	256
12.11 phx—Show ph Search Results in an X Window	257
Summary.....	257
■ Chapter 13: Home on the Web	259
Playing with Hypertext: The html-funcs Library	259
13.1 get_element—Extract the First Occurrence of an Element	259
13.2 split_tags—Put Each Tag on Its Own Line.....	262
13.3 html-title—Get the Title from an HTML File	263
HTML on the Fly: The cgi-funcs Library.....	265
13.4 x2d2—Convert a Two-Digit Hexadecimal Number to Decimal	265
13.5 dehext—Convert Hex Strings (%XX) to Characters	266
13.6 filedate—Find and Format the Modification Date of a File	268
Creating HTML Files	269
13.7 mk-htmlindex—Create an HTML Index	269
13.8 pretext—Create a Wrapper Around a Text File	276
13.9 text2html—Convert a Text File to HTML.....	277
13.10 demo.cgi—A CGI Script.....	278
Summary.....	280
■ Chapter 14: Taking Care of Business.....	281
14.1 prcalc—A Printing Calculator.....	281
How It Works.....	281
14.2 gle—Keeping Records Without a Shoebox	286
How It Works.....	286
Summary.....	296

■ Chapter 15: Random Acts of Scripting	297
The rand-funcs Library.....	297
15.1 random—Return One or More Random Integers in a Given Range	297
How It Works.....	298
Notes	301
15.2 toss—Simulate Tossing a Coin	301
How It Works.....	301
15.3 randstr—Select a String at Random	302
How It Works.....	302
A Random Sampling of Scripts.....	302
15.4 rand-date—Generate Random Dates in ISO Format.....	303
How It Works.....	303
Notes	304
15.5 randsort—Print Lines in Random Order.....	305
How It Works.....	305
15.6 randomword—Generate Random Words According to Format Specifications	306
How It Works.....	306
15.7 dice—Roll a Set of Dice	311
How It Works.....	311
15.8 throw—Throw a Pair of Dice.....	314
How It Works.....	314
Summary	315
■ Chapter 16: A Smorgasbord of Scripts	317
16.1 topntail—Remove Top and Bottom Lines from a File.....	317
How It Works.....	317
16.2 flocate—Locate Files by Filename Alone	319
How It Works.....	319
16.3 sus—Display a POSIX Man Page	320
How It Works.....	320

16.4 cwbw—Count Words Beginning With	321
How It Works.....	321
Notes	322
16.5 cci—Configure, Compile, and Install from Tarball	322
How It Works.....	322
Notes	323
16.6 ipaddr—Find a Computer’s Network Address.....	323
How It Works.....	324
16.7 ipaddr.cgi—Print the Remote Address of an HTTP Connection.....	324
How It Works.....	324
16.8 iprev—Reverse the Order of Digits in an IP Address	325
How It Works.....	325
Notes	325
16.9 intersperse—Insert a String Between Doubled Characters	326
How It Works.....	326
16.10 ll—Use a Pager for a Directory Listing Only If Necessary.....	327
How It Works.....	327
16.11 name-split—Divide a Person’s Full Name into First, Last, and Middle Names	328
How It Works.....	328
16.12 rot13—Encode or Decode Text	329
How It Works.....	329
Notes	330
16.13 showfstab—Show Information from /etc/fstab	330
How It Works.....	330
Notes	331
16.14 unique—Remove All Duplicate Lines from a File.....	331
How It Works.....	331
Summary	332

■ Chapter 17: Script Development Management	333
17.1 script-setup—Prepare the Scripting Environment.....	333
How It Works.....	333
Notes	337
17.2 cpsh—Install Script and Make Backup Copy	338
How It Works.....	338
17.3 shgrep—Search Scripts for String or Regular Expression.....	339
How It Works.....	340
17.4 shcat—Display a Shell Script	340
How It Works.....	340
Summary	341
■ Appendix A: Internet Scripting Resources	343
Introductions to Shell Scripting.....	343
Intermediate and Advanced Scripting	343
Collections of Scripts	344
Home Pages for Shells	344
Regular Expressions, sed, and awk.....	344
Miscellaneous Pages.....	344
History of the Shell	345
Index	347

About the Authors



Chris F. A. Johnson was introduced to Unix in 1990 and learned shell scripting because there was no C compiler on the system. His first major project was a menu-driven, user-extensible database system with report generator. Chris uses the shell as his primary, general-purpose programming language, and his projects have included a member database, menuing system, and POP3 mail filtering and retrieval. Chris is the coauthor of *Pro Bash Programming* (Apress, 2015). When not pushing shell scripting to the limit, he designs and codes web sites, teaches chess, and composes cryptic crosswords.



Jayant Varma is the founder of OZ Apps (www.oz-apps.com), a consulting, training, and development company providing IT solutions (specialization in mobile technology). He is an experienced developer with more than 20 years of industry experience spread across several countries. He is the author of a number of books on iOS development, including *Learn Lua for iOS Game Development* (Apress, 2012), *Xcode 6 Essentials* (Packt, 2015), *More iPhone Development with Swift* (Apress, 2015), and *More iPhone Development with Objective-C* (Apress, 2015). He has also been a university lecturer in Australia where he currently resides. He loves traveling and finds Europe to be his favorite destination.

Acknowledgments

I would like to thank the wonderful staff at Apress for the opportunity to update this book. Special thanks to Louise and Mark, who facilitated the quick turnaround on the book and getting it to print. Lastly, special thanks to my family for their support in getting this book completed.

—Jayant Varma

CHAPTER 1



The POSIX Shell and Command-Line Utilities

The POSIX shell is a descendant of the KornShell, which is a descendant of the Bourne shell. The basic syntax has remained the same, and Bourne shell scripts will usually run successfully in a POSIX shell. Not all of the KornShell features are included in POSIX (there are no arrays, for example), but the most important ones are. Those, which would be important to many developers, are string manipulation and arithmetic.

The scripts in this book make extensive use of features of the POSIX shell, and keep external commands to a minimum. This chapter presents an overview of the features of the POSIX shell and the external Unix commands used in this book, without going into great detail. Further information is available in the documentation for the shell and the various commands as well as on many web pages, a number of which are listed in the Appendix. We will also explain some of the idiosyncrasies used in the scripts, and present a library of functions that are used by many scripts.

Shell Commands

These descriptions are brief overviews of the built-in commands; for a complete description, see your shell's man page.

echo

The echo command prints its arguments separated by single spaces followed by a newline. If an unquoted variable contains characters present in `$IFS` (see “Parameters and Variables” later in this chapter), then the variable will be split on those characters:

```
$ list="a  b c d e f  g  h"
$ echo $list
a b c d e f g h
```

If the variable is quoted, all internal characters will be preserved:

```
$ echo "$list"
a  b c d e f  g  h
```

In the early days of Unix, two different versions of `echo` appeared. One version converted escape sequences, such as `\t` and `\n`, into the characters they represent in the C language; `\c` suppressed the newline, and discarded any further characters. The other used the `-n` option to suppress the trailing newline and did not convert escape sequences. The POSIX standard for `echo` says that “Implementations shall not support any options” and “If the first operand is `-n`, or if any of the operands contain a backslash (`\`) character, the results are implementation-defined.” In other words, you cannot rely on `echo`’s behavior being one or the other. It is best not to use `echo` unless you know exactly what `echo` is going to print, and you know that it will not contain any problem characters. The preferred command is `printf`.

printf

This command may be built into the shell itself, or it may be an external command. Like the C-language function on which it is based, `printf` takes a format operand that describes how the remaining arguments are to be printed, and any number of optional arguments. The format string may contain literal characters, escape sequences, and conversion specifiers. Escape sequences (the most common ones being `\n` for newline, `\t` for tab, and `\r` for carriage return) in format will be converted to their respective characters. Conversion specifiers, `%s`, `%b`, `%d`, `%x`, and `%o`, are replaced by the corresponding argument on the command line. Some implementations support other specifiers, but they are not used in this book. When there are more arguments than specifiers, the format string is reused until all the arguments have been consumed.

The `%s` specifier interprets its argument as a string and prints it literally:

```
$ printf "%s\n" "qwer\ty" 1234+5678
qwer\ty
1234+5678
```

The `%b` specifier is like `%s`, but converts escape sequences in the argument:

```
$ printf "%b\n" "qwer\ty" "asdf\nghj"
qwer      y
asdf
ghj
```

The `%d`, `%x`, and `%o` specifiers print their arguments as decimal, hexadecimal, and octal numbers, respectively.

```
$ printf "%d %x %o\n" 15 15 15
15 f 17
```

The conversion specifiers may be preceded by flags for width specification, optionally preceded by a minus sign indicating that the conversion is to be printed flush left, instead of flush right, in the specified number of columns:

```
$ printf "%7d:\n%7s:\n%-7s:\n" 23 Cord Auburn
      23:
    Cord:
Auburn :
```

In a numeric field, a 0 before the width flag indicates padding with zeroes:

```
$ printf "%07d\n" 13
0000013
```

set

In the *Oxford English Dictionary*, the longest entry is for the word *set*—thirty-two pages in my Compact Edition. In the Unix shell, the `set` command is really three commands in one. Without any arguments, it prints the names and values of all shell variables (including functions). With one or more option arguments, it alters the shell's behavior. Any non-option arguments are placed in the positional parameters.

Only three options to `set` are used in this book:

- `-v`: Print shell input lines as they are read.
- `-x`: Print commands and their arguments as they are executed.
- `-f`: Disable file name generation (globbing).

Given this script, which is call `xx.sh`,

```
echo "Number of positional parameters: $#"  
echo "First parameter: ${1:-EMPTY}"  
shift $(( $# - 1 ))  
echo "Last parameter: ${1:-EMPTY}"
```

its output is this:

```
$ xx.sh the quick brown fox  
Number of positional parameters: 4  
First parameter: the  
Last parameter: fox
```

If `set -v` is added to the top of the script, and the standard output redirected to oblivion, the script itself is printed:

```
$ xx.sh the quick brown fox >/dev/null  
echo "Number of positional parameters: $#"  
echo "First parameter: ${1:-EMPTY}"  
shift $(( $# - 1 ))  
echo "Last parameter: ${1:-EMPTY}"
```

If `set -v` is replaced with `set -x`, variables and arithmetic expressions are replaced by their values when the lines are printed; this is a useful debugging tool:

```
$ xx.sh the quick brown fox >/dev/null  
++ echo 'Number of positional parameters: 4'  
++ echo 'First parameter: the'  
++ shift 3  
++ echo 'Last parameter: fox'  
++ exit
```

To demonstrate the `set -f` option, and the use of `+` to reverse the operation, run the following script in an empty directory:

```
## Create a number of files using brace expansion (bash, ksh)
touch {a,b,c,d}${RANDOM}_{e,r,g,h}${RANDOM}

## Turn off filename expansion
set -f
printf "%-22s%-22s%-22s\n" * ; echo ## Display asterisk
printf "%-22s%-22s%-22s\n" *h* ; echo ## Display "*h*"

## Turn filename expansion back on
set +f
printf "%-22s%-22s%-22s\n" * ; echo ## Print all filenames
printf "%-22s%-22s%-22s\n" *h* ; echo ## Print filenames containing "h"
```

When the script is run, this is the output:

```
$ xx.sh
*

*h*

a12603_e6243          a28923_h23375          a29140_r28413
a5760_g7221           b17774_r4121           b18259_g11343
b18881_e10656         b660_h32228           c22841_r19358
c26906_h14133         c29993_g6498          c6576_e25837
d11453_h12972         d25162_e3276          d7984_r25591
d8972_g31551

a28923_h23375         b660_h32228           c26906_h14133
d11453_h12972
```

You can use `set` to split strings into pieces by changing the value of `$IFS`. For example, to split a date, which could be 2005-03-01 or 2003/09/29 or 2001.01.01, `$IFS` can be set to all the possible characters that could be used as separators. The shell will perform word splitting on any character contained in `$IFS`:

```
$ IFS=' -/.'
$ set 2005-03-01
$ printf "%s\n" "$@"
2005
03
01
```

When the value to be set is contained in a variable, a double dash should be used to ensure that the value of the variable is not taken to be an option:

```
$ var="-f -x -o"
$ set -- $var
```

shift

The leading positional parameters are removed, and the remaining parameters are moved up. By default, one parameter is removed, but an argument may specify more:

```
$ set 1 2 3 4 5 6 7 8
$ echo "$* ($#)"
1 2 3 4 5 6 7 8 (8)
$ shift
$ echo "$* ($#)"
2 3 4 5 6 7 8 (7)
$ shift 3
$ echo "$* ($#)"
5 6 7 8 (4)
```

■ **Tip** This is most useful when used in scripts to iterate through a number of parameters passed. Access the **\$0** parameter and keep shifting till there are no more to go through. However shift removes the parameters, so be aware if you want to reuse the parameters.

Some shells will complain if the argument to shift is larger than the number of positional parameters.

type

The POSIX standard says type “shall indicate how each argument would be interpreted if used as a command name.” Its return status may be used to determine whether a command is available:

```
if type stat > /dev/null 2>&1 ## discard the output
then
    stat "$file"
fi
```

If the command is an executable file, type prints the path to the file; otherwise, it prints the type of command that will be invoked: function, alias, or shell builtin. Its output is not standard across different shells, and therefore cannot be used reliably in a shell script.

The four arguments to type in the following example represent an executable file, a function, a nonexistent command, and an alias.

```
$ type ls pr1 greb ecoh
ls is hashed (/bin/ls)
pr1 is a function
pr1 ()
{
```

```

case $1 in
    -w)
        pr_w=
        ;;
    *)
        pr_w=-. ${COLUMNS:-80}
        ;;
esac;
printf "%${pr_w}s\n" "$@"
}
bash: type: greb: not found
echo is aliased to `echo`

```

Unlike most shells, bash will print the definition of a function.

getopts

The command `getopts` parses the positional parameters according to a string of acceptable options. If an option is followed by a colon, an argument is expected for that option, and will be stored in `$OPTARG`. This example accepts `-a`, `-b`, and `-c`, with `-b` expecting an argument:

```

while getopts ab:c opt
do

    case $opt in
        a) echo "Option -a found" ;;
        b) echo "Option -b found with argument $OPTARG" ;;
        c) echo "Option -c found" ;;
        *) echo "Invalid option: $opt"; exit 5 ;;
    esac
done

```

case

A workhorse among the shell's built-in commands, `case` allows multiple branches, and is the ideal tool, rather than `grep`, for determining whether a string contains a pattern or multiple patterns. The format is

```

case STRING in
    PATTERN [| PATTERN ...]) [list] ;;
    [PATTERN [| PATTERN ...]) [list] ;; ...]
esac

```

The `PATTERN` is a pathname expansion pattern, not a regular expression, and the list of commands following the first `PATTERN` that matches is executed. (See the “Patterns” section further on for an explanation of the two types of pattern matching.)

eval

The command `eval` causes the shell to evaluate the rest of the line, then execute the result. In other words, it makes two passes at the command line. For example, given the command:

```
eval "echo \${$#}"
```

The first pass will generate `echo ${4}` (assuming that there are 4 positional parameters). This will then print the value of the last positional parameter, `$4`.

local

The `local` command is used in functions; it takes one or more variables as arguments and makes those local to the function and its children. Though not part of the POSIX standard, it is built into many shells; `bash` and the `ash` family have it, and `pksh` has it as a standard alias for `typeset` (which is also not included in POSIX). In KornShell 93 (generally referred to as `ksh93`), if a function is defined in the portable manner (as used throughout this book), there is no way to make a variable local to a function.

In this book, `local` is used only in the few scripts that are written specifically for `bash`, most often for setting `IFS` without having to restore it to its original value:

```
local IFS=$NL
```

Parameters and Variables

Parameters are names used to represent information; there are three classes of parameters:

Positional parameters are the command-line arguments, and are numbered beginning with `$1`; variables are parameters denoted by a name that contains only letters, numbers and underscores, and that begins with a letter or an underscore; and special parameters that are represented by non-alphanumeric characters.

Positional Parameters

Positional parameters are the command-line arguments passed to a script or a function, and are numbered beginning with 1. Parameters greater than 9 must be enclosed in braces: `${12}`. This is to preserve compatibility with the Bourne shell, which could only access the first nine positional parameters; `$12` represents the contents of `$1`, followed by the number 2. The positional parameters can be assigned new values, with the `set` command. (See the example under “Special Parameters.”)

Special Parameters

The parameters `$*` and `$@` expand to all the positional parameters, and `#` represents the number of positional parameters. This function demonstrates the features of these parameters:

```
demo()
{
    printf "Number of parameters: %d\n" $#
    printf " The first parameter: %s\n" "$1"
    printf "The second parameter: %s\n" "$2"
    printf "\nAll the parameters, each on a separate line:\n"
    printf "\t%s\n" "$@"
}
```

```

printf "\nAll the parameters, on one line:\n"
printf "\t%s\n" "$*"
printf "\nEach word in the parameters on its own line:\n"
printf "\t%s\n" $*
}

```

Here, the demo function is run with three arguments:

```

$ demo The "quick brown" fox
Number of parameters: 3
  The first parameter: The
The second parameter: quick brown

```

```

All the parameters, each on a separate line:
  The
  quick brown
  fox

```

```

All the parameters, on one line:
  The quick brown fox

```

```

Each word in the parameters on its own line:
  The
  quick
  brown
  fox

```

The decimal exit code of the previous command executed (0 for success, non-zero for failure) is stored in \$?:

```

$ true; echo $?
0
$ false; echo $?
1

```

The shell's current option flags are stored in \$-; the shell's process ID is in \$\$; \$! is the process ID of the most recently executed background command, and \$0 is the name of the current shell or script:

```

$ sleep 4 &
[1] 12725
$ printf "PID: %d\nBackground command PID: %d\n" $$ $!
PID: 12532
Background command PID: 12725
$ printf "Currently executing %s with options: %s\n" "$0" "$-"
Currently executing bash with options: fhimBH

```

Shell Variables

These are the variables that are assigned values at the command line or in a script. The system or the shell itself also set a number of variables; those used in this book are

- `$HOME`: The path name of user's home directory (e.g., `/home/chris`).
- `$IFS`: A list of characters used as internal field separators for word splitting by the shell. The default characters are space, tab, and newline. Strings of characters can be broken up by changing the value of `$IFS`:

```
$ IFS=-; date=2005-04-11; printf "%s\n" $date
2005
04
11
```

- **\$PATH:** This colon-separated list of directories tells the shell which directories to search for a command. To execute a command in other directories, including the current working directory, an explicit path must be given (`/home/chris/demo_script` or `./demo_script`, not just `demo_script`).
- **\$PWD:** This is set by the shell to the pathname of the current working directory:

```
$ cd $HOME && echo $PWD
/home/chris
$ cd "$puzzles" && echo $PWD
/data/cryptics
```

standard-vars—A Collection of Useful Variables

My standard-vars file begins with these lines:

NL=''
'
CR=''
'
TAB=' '

You might be able to guess that these three variables represent newline, carriage return, and tab, but it's not clear, and cannot be cut and pasted from a web site or newsgroup posting. Once those variables are successfully assigned, however, they can be used, without ambiguity, to represent those characters. The `standard-vars` file is read by the shell and executed in the current environment (known as *sourcing*, it is described later in the chapter) in most of my shell scripts, usually via `standard-funcs`, which appears later in this chapter.

Create the file with the following script, then add other variables as we found them useful:

```
printf "%b\n" \
  "NL=\"\n\"" \
  "CR=\"\r\"" \
  "TAB=\"\t\"" \
  "ESC=\"\e\"" \
  "SPC=\"\040\"" \
  "export NL CR TAB ESC SPC" > $HOME/scripts/standard-vars.sh
```

The `-sh` extension is part of the system used for working on scripts without contaminating their production versions. It is explained in Chapter 20.