
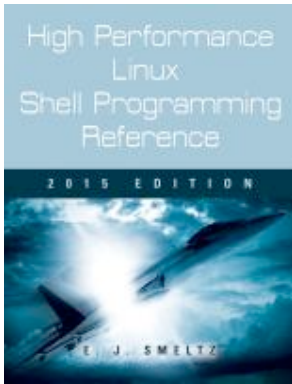


High Performance Linux Shell Programming Reference

2 0 1 5 E D I T I O N



E . J . S M E L T Z



Extensive, example-based Linux shell programming reference includes an English-to-shell dictionary, a tutorial and handbook, and many tables of information useful to programmers. Besides listing more than 2000 shell one-liners, it explains the principles and techniques of how to increase performance (execution speed, reliability, and efficiency), which apply to many other programming languages beyond shell.

High Performance Linux Shell Programming Reference 2015 Edition

Order the complete book from

Booklocker.com

<http://www.booklocker.com/p/books/7831.html?s=pdf>

or from your favorite neighborhood
or online bookstore.

Your free excerpt appears below. Enjoy!

**High
Performance
Linux Shell
Programming
Reference**

2015 Edition

High Performance Linux Shell Programming Reference, 2015 Edition
Copyright © 2015 by Edward J. Smeltz

ISBN 978-1-63263-401-6

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, recording or otherwise, without the prior written permission of the author.

Printed on acid-free paper

All information herein is believed to be accurate and correct, but the author and Booklocker.com, Inc assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained in this book.

Manufacturers and sellers often use specific designations for their products to distinguish them in the marketplace. Where such designations appear in this book, and E. J. Smeltz was aware of a trademark claim, the designations have been printed in all caps or in initial caps. All trademarks are the property of their respective owners.

Each URL cited herein was operational as of the time the passage referring to each was written. Because web sites sometimes relocate or disappear from the web, some web pages might cease to appear at the URLs indicated for them at a future point in time. A web search might reveal a new URL for the desired material if a cited URL goes away.

Booklocker.com, Inc. 2015
www.BookLocker.com

Watch for the upcoming novel series from E. J. Smeltz entitled *Obeying God Uphill*.

Table of Contents

Preface	xiii
Section 1 - Basic Information	1
1.01 Why Use Linux Shell Scripts?	3
1.02 The Structure of this Book	7
1.03 How to Create a Safe Environment for Learning Shell	13
1.04 How to Find Commands	17
1.05 Conventions and Terminology	21
1.06 Glossary of Shell Commands and Keywords of Interest	23
1.07 Common Linux System Status Commands & Tools	41
1.08 Commands for Delayed, Periodic, or Unattended Execution	43
1.09 Shells	45
1.10 bash Internal Shell Variables and set Options	49
1.11 File Types, File Extensions, and Magic Numbers	53
1.12 POSIX Character Class Definitions	59
1.13 ASCII Character Tables	63
1.14 ANSI Escape Codes	71
Section 2 - Linux Shell Programming Dictionary	75
2.01 Appending	83
2.02 Archiving, Unarchiving	91
2.03 Calculating General Numbers and Dates	101
2.04 Calculating Geometric Quantities	121
2.05 Calculating Maximums, Minimums, Means, Medians, Modes, and Totals	129
2.06 Comparing Files, Running Checksums	137
2.07 Compressing, Uncompressing	141
2.08 Converting Data from One Type to Another	157
2.09 Converting Numbers from One Base to Another	167
2.10 Copying and Duplicating Characters, Words, Fields, and Lines	171
2.11 Copying and Moving Files on a Single Host or Between Multiple Hosts	177
2.12 Counting, Indexing, Numbering, Serializing	203
2.13 Deleting Characters, Words, Fields, and Lines	213
2.14 Encoding, Decoding	233
2.15 Finding Files, Executing Commands on Found Files	237
2.16 Formatting Data, Reformatting Data	261
2.17 Generating Numbers and Strings, Random and Non-random	281
2.18 Graphing, Plotting Data	297
2.19 Inserting, Including	309
2.20 Joining Lines Horizontally, Pasting Files Side by Side	321
2.21 Joining Lines Vertically, Concatenating Files and Streams	325
2.22 Listing, Extracting, Displaying	329
2.23 Looping and Converting Between Pipelines and Variables	347
2.24 Prefixing, Prepending	373
2.25 Scheduling and Timing Job Execution	381
2.26 Separating Lines Horizontally	387

2.27	Separating, Splitting Lines Vertically.....	391
2.28	Shifting, Rearranging Items Horizontally Within a Line.....	395
2.29	Shuffling, Randomizing Line Order.....	399
2.30	Sorting Line Order.....	403
2.31	Sorting, Separating Items to Different Destinations.....	409
2.32	Substituting, Translating, Replacing One Item with Another.....	415
2.33	Testing, Conditionals, Exit Values, Pipelines as Conditionals.....	423
Section 3 - One-Liners that Show or Control the State of the Computer.....		457
3.01	CPU, Memory, Drivers, Kernel, Time, Locale, and General Hardware.....	459
3.02	Data Storage, File Systems, Individual Files.....	475
3.03	Network.....	495
3.04	Peripherals.....	505
3.05	Users and Groups.....	509
Section 4 Tutorial and Handbook -- How to Write Shell Scripts.....		517
4.01	ABCs of Designing and Writing a Shell Script.....	521
4.02	Arrays.....	527
4.03	Automating Tasks and Sensing the State of the Computer.....	531
4.04	Benchmarking Commands.....	551
4.05	Capturing and Reproducing Interactive Sessions.....	559
4.06	Cleaning up Temporary Files.....	561
4.07	Combining, Splitting, and Buffering Data Streams.....	565
4.08	Controlling Data Stream Flow in Pipes.....	571
4.09	Controlling Program Execution Flow.....	581
4.10	Delimiters, Data Field Tags, Data Extraction Tips.....	589
4.11	Embedding Commands Within Commands.....	601
4.12	Enhancing Script Reliability.....	609
4.13	Finding and Using Points of Reference in a File.....	623
4.14	Generating a Basic Formatted Report.....	629
4.15	Handling Interrupts.....	633
4.16	Manipulating Data in Binary Files and Streams.....	637
4.17	Maximizing Execution Speed by Conserving Computer Resources.....	649
4.18	Metacharacters: Characters with a Second Meaning.....	677
4.19	Redirecting I/O on Local and Remote Hosts.....	697
4.20	Regular Expressions.....	717
4.21	Requesting, Conditioning and Qualifying User Inputs.....	737
4.22	Setting up a Working Environment from Within a Script.....	751
4.23	Simplifying Your Scripts.....	759
4.24	Standardizing Your Scripts.....	765
4.25	Table-driven Scripts.....	767
4.26	Thinking About Thinking: Engaging in Mental Gymnastics.....	771
4.27	Throttling and Overdrive Techniques.....	781
4.28	Zone 42: Interesting Solutions Looking for a Problem to Solve.....	789
Bibliography.....		797

1.07 Common Linux System Status Commands & Tools

General-purpose	RAM / Swap	Disk Hardware	Network Config	Hardware
atop	free	blockdev	ethtool	biosdecode
collectl	htop	hdparm	ifconfig	dmidecode
colplot	ipcs	lsblk	ip	lsdev
dmesg	numastat	lsscsi	ip6tables	lsdvb
dstat	slabtop	multipath	iptables	lshal
gdb	swapon		netstat	lshw
monit	top	Disk Partitions		lspci
nagios	vmstat	blkid	Network Diags	lspcmcia
nmon		cfdisk	arp	lsusb
oprofiled	Drivers / Modules	disktype	arping	mcelog
perf	ethtool	fdisk	fping	
sar	lsdev	gdisk	ifstat	Printing
stap	lsmod	kpartx	iftop	lp
valgrind	lspci	parted	iostat	lpadmin
xosview	modinfo	partx	iptraf	lpc
	modprobe	sfdisk	iptstate	lpinfo
CPU Hardware	systool	vmstat	mtr	lpoptions
arch			nload	lpq
lscpu	Users / Groups	LVM	ping	lpstat
uname	chage	blkid	tcptraceroute	
	finger	lvdisplay	traceroute	Display / Terminal
CPU Processes	groups	lvs		stty
jobs	grpck	pvdisk	Network Traffic	toe
pmap	last	pvs	argus	xlsclients
powertop	lastb	vgdisplay	cacti	xlsfonts
prtstat	pwck	vgs	dropwatch	
ps	w		dsniff	Misc
runlevel	who	File Systems	dumpcap	apachetop
strace	whoami	blkid	nmap	setserial
w	whois	df	ntop	stty
		du	p0f	
CPU Load	Files / Directories	dumpe2fs	tcpdump	NFS
htop	chkrootkit	dumpe4fs	vnstat	mountstats
iostat	cksum	findfs	wireshark	nfsiostat
mpstat	file	fuser	xprobe2	nfsstat
top	find	iostat		showmount
uptime	getfacl	iotop	Network Ports	
vmstat	ldd	lsof	lsof	Samba
	less	mount	netstat	net
Automatic	ls	stat	nc	nmblookup
at	lsattr	tune2fs	rpcinfo	smbclient
atq	lsinitrd	tune4fs	ss	smbstatus
chkconfig	lsof	vmstat		testparm
crontab	md5sum		DNS / Resolver	
service	more	Security	dig	Time
	objdump	certwatch	dnstop	date
Virtual machines	size	getenforce	getent	ntpdc
virsh	tree	getsebool	host	ntpq
virt-top	type	openssl	hostname	ntpstat
virt-what	wc	semanage	nslookup	ntptrace

The above commands are designed to show something about the computer system itself as a whole or the items in it: the hardware of which it is composed, the way it is configured, the things it is doing, the files in its file systems. And so on. Because they give a view into the local computer system or perhaps a remote host, they can be useful when you are troubleshooting a system or the things it is trying to do. Many of these commands overlap with one another or belong in more categories than what have been shown. One page holds only so much.

Some of these commands are strictly interactive, but many can be used in scripts. Some are installed as part of a base Linux system, but others must be specifically added. Not all commands exist on all varieties of Linux. This table is offered only as a general reference and is not by any means exhaustive. The table is an initial starting point of a place to look for commands that could be useful in discovering computer system or network status.

2.03 Calculating General Numbers and Dates

What This Chapter Covers

To "calculate" in this context means to use one or more mathematical operations to determine a numeric value. Examples of this are adding, subtracting, multiplying, dividing, and computing squares and square roots. Both integer and floating-point techniques are included. Data stream and shell variable approaches are covered.

This chapter covers the following types of calculations:

- integer addition, subtraction, multiplication, division, modulus (remainder)
- incrementing and decrementing
- truncating or rounding decimal numbers to integers
- showing absolute values
- calculating with numbers in bases other than base 10
- floating-point addition, subtraction, multiplication, division
- powers of n
- square roots
- calculating the value of e, natural logs and powers of e
- Bessel functions
- calculating the value of pi
- converting between degrees and radians
- outputting integer values
- outputting in floating-point format
- outputting in exponential notation
- outputting in exponential or floating-point (auto-choose)
- trigonometric functions: sin, cos, sec, csc, tan, cot
- trigonometric functions: arcsin, arccos, arctan
- hyperbolic functions: sinh, cosh, sech, csch, tanh, coth
- hyperbolic functions: arcsinh, arccosh, arctanh
- prime factors
- non-prime numbers
- prime numbers
- factorials
- number sequences (arithmetic and geometric)
- network settings (subnet masks, prefix bits, broadcast networks)
- dates, times, durations, calendars

Where to Find Other Things

To create random numbers, see the "Generating Numbers and Strings, Random and Non-random" chapter in Section 2. To convert between number bases, see the "Converting Numbers from One Base to Another" chapter in Section 2. To calculate perimeters, areas, and volumes, see the "Calculating Geometric Quantities" chapter in Section 2. To make statistical calculations, see the "Calculating Maximums, Minimums, Means, Medians, and Totals" chapter in Section 2. If you want to focus on counting, see the

"Counting, Indexing, Numbering, Serializing" chapter in Section 2 or the "Looping and Converting Between Pipelines and Variables" chapter in Section 2.

Warnings, Tips, and Suggestions

It is always good practice to use the `man` or `info` command (whichever is available) to verify the syntax and effect of any command before executing it on your computer.

Below are the meanings of the I/O identifiers:

- "[NN]" is "no data in, no data out": execute this as a standalone command line.
- "[NO]" is "no data in, data out": execute this to start a data stream.
- "[IO]" is "data in, data out": execute this in the middle of a data stream.
- "[IN]" is "data in, no data out": execute this to end a data stream.

The I/O identifier appears in the description portion of each one-liner table entry.

When you need to make floating-point calculations, I suggest you use commands such as `awk` or `bc` (binary calculator) to perform the calculations. Most other calculation methods on a Linux computer work only with integers. The `dc` command (desk calculator) provides another floating-point calculation method, but it uses reverse polish notation, and I do not care much for it. It seems unnecessarily confusing to use relative to the other computational methods. A few examples of `dc` are provided for the sake of completeness, but I prefer to deal with `awk` and `bc` instead.

Because of the importance of numeric calculations in shell scripting, I have tried in most cases in this chapter to include two types of one-liners for each operation: one type that processes data streams and one type that works with shell variables. As always, if you can process data as a stream instead of as individual shell variables, stream processing will tend to execute much faster. With the means to calculate using either approach, though, you can use the option that best fits your situation. Sometimes the nature of the task does not lend itself to stream-oriented computation, or if you execute a given command only once in each running of a script, execution speed might not matter much for that particular calculation.

In most chapters in Section 2, spaces are used in the one-liners to improve readability in places where they do not create a problem with syntax. In those other chapters, spaces often appear where they are allowed but not required. In this chapter, though, I make every effort to omit the spaces where they are not needed. This to help you get accustomed to seeing spaces where there need to be spaces. A one-liner such as

```
x=`expr $a + $b`
```

works fine as written, but if you omit the spaces around the plus sign as in

```
x=`expr $a+$b`
```

the one-liner breaks. Furthermore, the `expr` command in particular has some unusual character escaping needs, such as

```
x=`expr $a \* $b`
```

Therefore, if you try to execute

```
x=`expr $a * $b`
```

it will not work.

One-Liners: Calculating General Numbers and Time

Desired Action	Command Line
Integer addition, subtraction, multiplication, division, exponent, modulus (remainder)	
show the sum of field1 and field2: [IO]	<code>awk '{print int(\$1+\$2)}'</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=\$((\$a+\$b))</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=\$((\$a+\$b))</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=\$((expr \$a + \$b))</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=`expr \$a + \$b`</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>let x=\$((\$a+\$b))</code>
show the difference of field1 and field2: [IO]	<code>awk '{print int(\$1-\$2)}'</code>
put the difference of \$a and \$b into variable "x": [NN]	<code>x=\$((\$a-\$b))</code>
put the difference of \$a and \$b into variable "x": [NN]	<code>x=\$((\$a-\$b))</code>
put the difference of \$a and \$b into variable "x": [NN]	<code>x=\$((expr \$a - \$b))</code>
put the difference of \$a and \$b into variable "x": [NN]	<code>x=`expr \$a - \$b`</code>
put the difference of \$a and \$b into variable "x": [NN]	<code>let x=\$((\$a-\$b))</code>
show the product of field1 and field2: [IO]	<code>awk '{print int(\$1*\$2)}'</code>
put the product of \$a and \$b into variable "x": [NN]	<code>x=\$((\$a*\$b))</code>
put the product of \$a and \$b into variable "x": [NN]	<code>x=\$((\$a*\$b))</code>
put the product of \$a and \$b into variable "x": [NN]	<code>x=\$((expr \$a * \$b))</code>
put the product of \$a and \$b into variable "x": [NN]	<code>x=`expr \$a * \$b`</code>
put the product of \$a and \$b into variable "x": [NN]	<code>let x=\$((\$a*\$b))</code>
show field1 exponent field2: [IO]	<code>awk '{print int(\$1**\$2)}'</code>
show field1 exponent field2: [IO]	<code>awk '{print int(\$1^\$2)}'</code>

Desired Action	Command Line
put \$a exponent \$b into variable "x": [NN]	<code>x=\$((\$a**\$b))</code>
put \$a exponent \$b into variable "x": [NN]	<code>x=\$((\$a**\$b))</code>
put \$a exponent \$b into variable "x": [NN]	<code>let x=\$a**\$b</code>
show the value of field1 divided field2: [IO]	<code>awk '{print int(\$1/\$2)}'</code>
put the value of \$a divided by \$b into variable "x": [NN]	<code>x=\$((\$a/\$b))</code>
put the value of \$a divided by \$b into variable "x": [NN]	<code>x=\$((\$a/\$b))</code>
put the value of \$a divided by \$b into variable "x": [NN]	<code>x=\$((expr \$a / \$b))</code>
put the value of \$a divided by \$b into variable "x": [NN]	<code>x=`expr \$a / \$b`</code>
put the value of \$a divided by \$b into variable "x": [NN]	<code>let x=\$a/\$b</code>
show the remainder left when field1 is divided by field2: [IO]	<code>awk '{print \$1%\$2}'</code>
show the remainder left when \$a is divided by \$b: [NO]	<code>echo "\$a % \$b" bc</code>
show the integer remainder left when field1 is divided by field2: [IO]	<code>awk '{print int(\$1%\$2)}'</code>
put the remainder left when \$a is divided by \$b into variable "x": [NN]	<code>x=\$((\$a%\$b))</code>
put the remainder left when \$a is divided by \$b into variable "x": [NN]	<code>x=\$((expr \$a % \$b))</code>
put the remainder left when \$a is divided by \$b into variable "x": [NN]	<code>x=`expr \$a % \$b`</code>
put the remainder left when \$a is divided by \$b into variable "x": [NN]	<code>let x=\$a%\$b</code>
Incrementing and decrementing	
increment \$i: [IO]	<code>awk '{i=i+1;print i}'</code>
increment \$i: [NN]	<code>i=\$((\$i+1))</code>
increment \$i: [NN]	<code>i=\$((\$i+1))</code>
increment \$i: [NN]	<code>let i=\$i+1</code>
preincrement \$i: [IO]	<code>awk '{++i;print i}'</code>
preincrement \$i: [NN]	<code>i=\$((++i))</code>
postincrement \$i: [IO]	<code>awk '{i++;print i}'</code>
decrement \$i: [IO]	<code>awk '{i=i-1;print i}'</code>
decrement \$i: [NN]	<code>i=\$((\$i-1))</code>
decrement \$i: [NN]	<code>i=\$((\$i-1))</code>

Desired Action	Command Line
decrement \$i: [NN]	let i=\$((i-1))
predecrement \$i: [IO]	awk '{--i;print i}'
predecrement \$i: [NN]	i=\$((--i))
postdecrement \$i: [IO]	awk '{i--;print i}'
Truncating or rounding decimal numbers to an integer	
truncate field1 to an integer: [IO]	awk '{print int(\$1)}'
truncate field1 to an integer: [IO]	awk '{printf "%d\n",\$1}'
round field1 to the closer integer: [IO]	awk '(\$1-int(\$1))>=0.5 {print int(\$1+.5);next} {print int(\$1)}'
truncate variable \$x to an integer: [NN]	x=`awk -v x=\$x 'BEGIN {print int(x)}'`
truncate variable \$x to an integer: [NN]	x=`awk -v x=\$x 'BEGIN {printf "%d\n",x}'`
truncate variable \$x to an integer: [NN]	x=`echo \$x cut -d. -f1` Note: this does not handle 0.xxx or exponential numbers
truncate variable \$x to an integer: [NN]	x=`echo \$x sed '/\..*\$//'` Note: this does not handle 0.xxx or exponential numbers
truncate variable \$x to an integer: [NN]	x=`echo \$x sed '/\.[0-9]*\$//'` Note: this does not handle 0.xxx or exponential numbers
round variable \$x to the closer integer: [NN]	x=`awk -v x=\$x 'BEGIN {if (x-int(x)>=0.5){print int(x+.5)}else{print int(x)}}'`
Absolute values	
show the absolute value of field1: [IO]	awk '\$1 < 0 {print \$1*-1; next} {print \$1}'
show every field after converting field2 to its absolute value: [IO]	awk '\$2 < 0 {\$2=-\$2; print \$0; next} {print \$0}'
show the absolute value of every field: [IO]	awk '{for (f=1;f<=NF;f++) if (\$f<0) \$f=-\$f; print \$0}'
put the absolute value of variable "\$a" into variable "x": [NN]	x=`awk -v a=\$a \ 'BEGIN {if (a<0) a=-a; print a}'`
Calculating with numbers in bases other than base 10	
show in base 10 the product of base 2 values in field1 and field2: [IO]	awk 'NR==1{print "ibase=2"} {print \$1*\$2}' bc
put the product of base 2 numbers \$a and \$b into base 10 variable "x": [NN]	x=`echo "ibase=2;\$a*\$b" bc`
show in base 2 the product of base 10 values in field1 and field2: [IO]	awk 'NR==1{print "obase=2"} {print \$1*\$2}' bc
put the product of base 10 numbers \$a and \$b into base 2 variable "x": [NN]	x=`echo "obase=2;\$a*\$b" bc`
show in base 2 the product of base 2 values in field1 and field2: [IO]	awk 'NR==1{print "ibase=2"}{print \$1*\$2}' bc awk 'NR==1 {print "obase=2"}{print \$1}' bc

Desired Action	Command Line
put the product of base 2 numbers \$a and \$b into base 2 variable "x": [NN]	<code>x=`echo "ibase=2;\$a*\$b" bc awk '{print "obase=2;"\$1}' bc`</code>
show in base 10 the sum of base 8 values in field1 and field2: [IO]	<code>awk 'NR==1{print "ibase=8"}{print \$1+"\$2}' bc</code>
put the sum of base 8 numbers \$a and \$b into base 10 variable "x": [NN]	<code>x=`echo "ibase=8;\$a+\$b" bc`</code>
show in base 8 the sum of base 10 values in field1 and field2: [IO]	<code>awk 'NR==1{print "obase=8"}{print \$1+"\$2}' bc</code>
put the sum of base 10 numbers \$a and \$b into base 8 variable "x": [NN]	<code>x=`echo "obase=8;\$a+\$b" bc`</code>
show in base 8 the sum of base 8 values in field1 and field2: [IO]	<code>awk 'NR==1{print "ibase=8"}{print \$1+"\$2}' bc awk 'NR==1{print "obase=8"}{print \$1}' bc</code>
put the sum of base 8 numbers \$a and \$b into base 8 variable "x": [NN]	<code>x=`echo "ibase=8;\$a+\$b" bc awk '{print "obase=8;"\$1}' bc`</code>
show in base 10 the difference of base 16 values in field1 and field2: [IO]	<code>awk 'NR==1{print "ibase=16"}{print \$1-"\$2}' bc</code>
put the difference of base 16 numbers \$a and \$b into base 10 variable "x": [NN]	<code>x=`echo "ibase=16;\$a-\$b" bc`</code>
show in base 16 the difference of base 10 values in field1 and field2: [IO]	<code>awk 'NR==1{print "obase=16"}{print \$1-"\$2}' bc</code>
put the difference of base 10 numbers \$a and \$b into base 16 variable "x": [NN]	<code>x=`echo "obase=16;\$a-\$b" bc`</code>
show in base 16 the difference of base 16 values in field1 and field2: [IO]	<code>awk 'NR==1{print "ibase=16"}{print \$1-"\$2}' bc awk 'NR==1{print "obase=16"}{print \$1}' bc</code>
put the difference of base 16 numbers \$a and \$b into base 16 variable "x": [NN]	<code>x=`echo "ibase=16;\$a-\$b" bc awk '{print "obase=16;"\$1}' bc`</code>
Floating-point addition, subtraction, multiplication, division	
show the sum of field1 and field2: [IO]	<code>awk '{print (\$1+\$2)}'</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=`echo "scale=5;\$a+\$b" bc`</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {print (a+b)}'`</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=`echo "\$a \$b" awk '{print (\$1+\$2)}'`</code>
put the sum of \$a and \$b into variable "x": [NN]	<code>x=`echo "\$a \$b + p" dc`</code>
show the difference of field1 and field2: [IO]	<code>awk '{print (\$1-\$2)}'</code>

Desired Action	Command Line
put the difference of \$a and \$b into variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {print (a-b)}'`</code>
put the difference of \$a and \$b into variable "x": [NN]	<code>x=`echo "scale=5;\$a-\$b" bc`</code>
put the difference of \$a and \$b into variable "x": [NN]	<code>x=`echo "\$a \$b - p" dc`</code>
show the product of field1 and field2: [IO]	<code>awk '{print (\$1*\$2)}'</code>
put the product of \$a and \$b into variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b \ 'BEGIN {print (a*b)}'`</code>
put the product of \$a and \$b into variable "x": [NN]	<code>x=`echo "scale=5;\$a*\$b" bc`</code>
put the product of \$a and \$b into variable "x": [NN]	<code>x=`echo "\$a \$b * p" dc`</code>
show field1 divided by field2: [IO]	<code>awk '{print (\$1/\$2)}'</code>
put the result of \$a divided by \$b into variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b \ 'BEGIN {print (a/b)}'`</code>
put the result of \$a divided by \$b into variable "x": [NN]	<code>x=`echo "scale=5;\$a/\$b" bc`</code>
Powers of n	
show field1 raised to the power of field2: [IO]	<code>awk '{print (\$1**\$2)}'</code>
show field1 raised to the power of field2: [IO]	<code>awk '{print (\$1^\$2)}'</code>
put the result of \$a raised to the power of \$b into variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b \ 'BEGIN {print (a^b)}'`</code>
put the result of \$a raised to the power of \$b into variable "x": [NN]	<code>x=`echo "scale=5;\$a^\$b" bc`</code>
put the result of \$a raised to the power of \$b into variable "x": [NN]	<code>x=\$((\$a**\$b))</code>
put the result of \$a raised to the power of \$b into variable "x": [NN]	<code>x=\$((\$a**\$b))</code>
Square roots	
show the square root of field1: [IO]	<code>awk '{print sqrt(\$1)}'</code>
put the square root of \$a into variable "x": [NN]	<code>x=`awk -v a=\$a 'BEGIN {print sqrt(a)}'`</code>
put the square root of \$a into variable "x": [NN]	<code>x=`echo "scale=5;sqrt(\$a)" bc`</code>
Calculating the value of e, natural logs and powers of e	In my testing, bc correctly calculated e to beyond 200 digits. But awk was accurate to 14 digits. Verify how your system behaves.
show the value of e: [NO]	<code>awk 'BEGIN{print exp(1)}'</code>

Desired Action	Command Line
show the value of e: [NO]	<code>echo "e(1)" bc -l</code>
show the value of e to 14 digits: [NO]	<code>awk 'BEGIN{printf "%.14f\n",exp(1)}'</code>
show the value of e to 14 digits: [NO]	<code>echo "scale=14;e(1)" bc -l</code>
show the value of e to 200 digits: [NO]	<code>echo "scale=200;e(1)" bc -l</code>
show the natural log of field1: [IO]	<code>awk '{print log(\$1)}'</code>
show the natural log of field1 to 40 digits: [IO]	<code>awk 'NR<2{print "scale=40"}{print "l("\$1)"}' bc -l</code>
put the natural log of \$a into variable "x": [NN]	<code>x=`awk -v a=\$a 'BEGIN {print log(a)}'`</code>
put the natural log of \$a into variable "x": [NN]	<code>x=`echo "scale=5;l(\$a)" bc -l`</code>
show e^{field1} : [IO]	<code>awk '{print exp(\$1)}'</code>
show e^{field1} to 40 digits: [IO]	<code>awk 'NR<2{print "scale=40"}{print "e("\$1)"}' bc -l</code>
put $e^{\text{\$a}}$ into variable "x": [NN]	<code>x=`awk -v a=\$a 'BEGIN {print exp(a)}'`</code>
put $e^{\text{\$a}}$ into variable "x": [NN]	<code>x=`echo "scale=5;e(\$a)" bc -l`</code>
Bessel functions	
show the Bessel function of integer order field1 of field2: [IO]	<code>awk 'NR==1{print "scale=5"}{"j("\$1","\$2)"}' bc -l</code>
put the Bessel function of integer order \$a of \$b into variable "x": [NN]	<code>x=`echo"scale=5;j(\$a,\$b)" bc -l`</code>
Calculating the value of pi	
Note: if you require the value of pi to an extended number of decimal places, look up the value and hard-code it into the script. Then verify that your chosen math operators can provide the correct answer based on that many digits of precision.	Note: just because you tell something to compute the value of pi to so many digits does not mean those computed digits will be accurate. In my testing, awk was accurate to 14 digits, but bc was accurate to 37. Verify how your computer behaves.
show the value of pi: [NO]	<code>awk 'BEGIN {print atan2(0,-1)}'</code>
show the value of pi: [NO]	<code>awk 'BEGIN {print (4*atan2(1,1))}'</code>
show the value of pi to 14 digits: [NO]	<code>awk 'BEGIN {printf "%.14f\n",atan2(0,-1)}'</code>
show the value of pi to 14 digits: [NO]	<code>awk 'BEGIN {printf "%.14f\n", (4*atan2(1,1))}'</code>
show the value of pi to 14 digits: [NO]	<code>echo "scale=14;4*a(1)" bc -l</code>
show the value of pi to 37 digits: [NO]	<code>echo "scale=37;4*a(1)" bc -l</code>
put the value of pi to 10 digits into the variable "pi": [NN]	<code>pi=\$(echo "scale=10;4*a(1)" bc -l)</code>
put the value of pi to 10 digits into the variable "pi": [NN]	<code>pi=\$(awk 'BEGIN {printf "%.10f\n",atan2(0,-1)}')</code>
put the value of pi to 10 digits into the variable "pi": [NN]	<code>pi=\$(awk 'BEGIN {printf "%.10f\n", (4*atan2(1,1))}'')</code>
put the value of pi to 10 digits into the variable "pi": [NN]	<code>pi=\$(awk 'BEGIN {print "scale=10;4*a(1)" "bc -l"}')</code>

Desired Action	Command Line
put the value of pi to 37 digits into the variable "pi": [NN]	<code>pi=\$(echo "scale=37;4*a(1)" bc -l)</code>
Converting between degrees and radians	
convert degrees in field1 to radians to 10 digits: [IO]	<code>awk '{printf "%.10f\n", \(\$1/45*atan2(1,1))}'</code>
convert degrees in field1 to radians to 10 digits: [IO]	<code>awk 'NR==1{print "scale=10"} {print \$1"/45*a(1)"}' bc -l</code>
convert radians in field1 to degrees to 10 digits: [IO]	<code>awk '{printf "%.10f\n", \(\$1*45/atan2(1,1))}'</code>
convert radians in field1 to degrees to 10 digits: [IO]	<code>awk 'NR==1{print "scale=10"} {print \$1"*45*a(1)"}' bc -l</code>
put the number of degrees per radian to 10 digits into the variable "dpr": [NN]	<code>dpr=`echo "scale=10;45/a(1)" bc -l`</code>
put the number of radians per degree to 10 digits into the variable "rpd": [NN]	<code>rpd=`echo "scale=10;a(1)/45" bc -l`</code>
Outputting integer values	
show the integer value of field1 / field2: [IO]	<code>awk '{printf "%d\n", \$1/\$2}'</code>
show the integer value of field1 / field2 (global format affects all numbers output): [IO]	<code>awk 'BEGIN {OFMT="%d"} {print \$1/\$2}'</code>
put the integer value of \$a / \$b into the variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {printf "%d\n", a/b}'`</code>
Outputting in floating-point format	
put the value of \$a / \$b to 5 digits into the variable "x": [NN]	<code>x=`echo "scale=5;\$a/\$b" bc`</code>
put the value of \$a / \$b to 20 digits into the variable "x": [NN]	<code>x=`echo "scale=20;\$a/\$b" bc`</code>
show the value of field1 / field2 to 5 digits: [IO]	<code>awk '{printf "%.5f\n", \$1/\$2}'</code>
show the value of field1 / field2 to 5 digits (global format affects all numbers output): [IO]	<code>awk 'BEGIN {OFMT="%.5f"} {print \$1/\$2}'</code>
put the value of \$a / \$b to 5 digits into the variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {printf "%.5f\n", a/b}'`</code>
put the value of \$a / \$b to 5 digits into the variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {OFMT="%.5f"} {print a/b}'`</code>
Outputting in exponential notation	

Desired Action	Command Line
show the value of field1 / field2 to 5 digits in exponential notation: [IO]	<code>awk '{printf "%.5e\n", \$1/\$2}'</code>
show the value of field1 / field2 to 5 digits in exponential notation (global format affects all numbers output): [IO]	<code>awk 'BEGIN {OFMT="%.5e"} {print \$1/\$2}'</code>
put the value of \$a / \$b to 5 digits in exponential notation into the variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {printf "%.5e\n", a/b}'`</code>
Outputting in exponential or floating-point (auto-choose)	
show the value of field1 / field2 to 5 digits in exponential or floating-point notation: [IO]	<code>awk '{printf "%.5g\n", \$1/\$2}'</code>
show the value of field1 / field2 to 5 digits in exponential or floating-point notation (global format affects all numbers output): [IO]	<code>awk 'BEGIN {OFMT="%.5g"} {print \$1/\$2}'</code>
put the value of \$a / \$b to 5 digits in exponential or floating-point notation into the variable "x": [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {printf "%.5g\n", a/b}'`</code>
Trigonometric functions: sin, cos, sec, csc, tan, cot	
show the sine of field1 (input value is in degrees): [IO]	<code>awk '{print sin(\$1/45*atan2(1,1))}'</code>
show the sine of field1 (input value is in radians): [IO]	<code>awk '{print sin(\$1)}'</code>
show the sine of field1 (input value is in radians): [IO]	<code>awk 'NR==1{print "scale=5"} {print "s("\$1)"}' bc -l</code>
put the sine of \$a into variable 'x' (input value is in radians): [NN]	<code>x=`awk -v a=\$a 'BEGIN {print sin(a)}'`</code>
put the sine of \$a into variable 'x' (input value is in radians): [NN]	<code>x=`echo "scale=5;s(\$a)" bc -l`</code>
show the cosine of field1 (input value is in degrees): [IO]	<code>awk '{print cos(\$1/45*atan2(1,1))}'</code>
show the cosine of field1 (input value is in radians): [IO]	<code>awk '{print cos(\$1)}'</code>
show the cosine of field1 (input value is in radians): [IO]	<code>awk 'NR==1{print "scale=5"} {print "c("\$1)"}' bc -l</code>
put the cosine of \$a into variable "x" (input value is in radians): [NN]	<code>x=`awk -v a=\$a 'BEGIN {print cos(a)}'`</code>
put the cosine of \$a into variable "x" (input value is in radians): [NN]	<code>x=`echo "scale=5;c(\$a)" bc -l`</code>

Desired Action	Command Line
show the secant of field1 (input value is in degrees): [IO]	<code>awk '{print 1/cos(\$1/45*atan2(1,1))}'</code>
show the secant of field1 (input value is in radians): [IO]	<code>awk '{print 1/cos(\$1)}'</code>
show the cosecant of field1 (input value is in degrees): [IO]	<code>awk '{print 1/sin(\$1/45*atan2(1,1))}'</code>
show the cosecant of field1 (input value is in radians): [IO]	<code>awk '{print 1/sin(\$1)}'</code>
show the tangent of field1 (input value is in radians): [IO]	<code>awk '{print (sin(\$1)/cos(\$1))}'</code>
put the tangent of \$a into variable "x" (input value is in radians): [NN]	<code>x=`echo "scale=5;s(\$a)/c(\$a)" bc -l`</code>
show the cotangent of field1 (input value is in radians): [IO]	<code>awk '{print 1/((sin(\$1)/cos(\$1))}'</code>
Trigonometric functions: arcsin, arccos, arctan, arccot	
show the arcsine of field1 (output value is in radians): [IO]	<code>awk '{print atan2(\$1,sqrt(1-\$1^2))}'</code>
show the arccosine of field1 (output value is in radians): [IO]	<code>awk '{print atan2(sqrt(1-\$1^2),\$1)}'</code>
show the arctangent of field1 / field2 (output value is in radians): [IO]	<code>awk '{print atan2(\$1,\$2)}'</code>
show the arctangent of field1 (output value is in radians): [IO]	<code>awk 'NR==1{print "scale=5"}{print "a("\$1)" }' bc -l</code>
put the arctangent of \$a / \$b into variable "x" (output value is in radians): [NN]	<code>x=`awk -v a=\$a -v b=\$b 'BEGIN {print atan2(a,b)}'`</code>
put the arctangent of \$a into variable "x" (output value is in radians): [NN]	<code>x=`echo "scale=5;a(\$a)" bc -l`</code>
show the arccotangent of field1 (output value is in radians): [IO]	<code>awk '{print (atan2(0,-1)/2)-atan2(\$1,1)}'</code>
Hyperbolic functions: sinh, cosh, sech, csch, tanh, coth	
show the hyperbolic sine of field1 (input value is in radians): [IO]	<code>awk '{print (exp(\$1)-exp(-\$1))/2}'</code>
put the hyperbolic sine of \$a into variable 'x' (input value is in radians): [NN]	<code>x=\$(echo "scale=5;(e(\$a)-e(-\$a))/2" bc -l)</code>
show the hyperbolic cosine of field1 (input value is in radians): [IO]	<code>awk '{print (exp(\$1)+exp(-\$1))/2}'</code>
put the hyperbolic cosine of \$a into variable 'x' (input value is in radians): [NN]	<code>x=\$(echo "scale=5;(e(\$a)+e(-\$a))/2" bc -l)</code>

Desired Action	Command Line
show the hyperbolic secant of field1 (input value is in radians): [IO]	<code>awk '{print 2/(exp(\$1)+exp(-\$1))}'</code>
put the hyperbolic secant of \$a into variable 'x' (input value is in radians): [NN]	<code>x=`echo "scale=5;2/(e(\$a)+e(-\$a))" bc -l`</code>
show the hyperbolic cosecant of field1 (input value is in radians): [IO]	<code>awk '{print 2/(exp(\$1)-exp(-\$1))}'</code>
put the hyperbolic cosecant of \$a into variable 'x' (input value is in radians): [NN]	<code>x=`echo "scale=5;2/(e(\$a)-e(-\$a))" bc -l`</code>
show the hyperbolic tangent of field1 (input value is in radians): [IO]	<code>awk '{print (exp(\$1)-exp(-\$1))/(exp(\$1)+exp(-\$1))}'</code> Note: type the above on one line.
put the hyperbolic tangent of \$a into variable 'x' (input value is in radians): [NN]	<code>x=\$(echo "scale=5;(e(\$a)-e(-\$a))/(e(\$a)+e(-\$a))" bc -l)</code> Note: type the above on one line.
show the hyperbolic cotangent of field1 (input value is in radians): [IO]	<code>awk '{print (exp(\$1)+exp(-\$1))/(exp(\$1)-exp(-\$1))}'</code> Note: type the above on one line.
put the hyperbolic cotangent of \$a into variable 'x' (input value is in radians): [NN]	<code>x=\$(echo "scale=5;(e(\$a)+e(-\$a))/(e(\$a)-e(-\$a))" bc -l)</code> Note: type the above on one line.
Hyperbolic functions: arcsinh, arccosh, arctanh	
show the hyperbolic arcsine of field1 (output value is in radians): [IO]	<code>awk '{print log(\$1+sqrt(\$1*\$1+1))}'</code>
show the hyperbolic arccosine of field1 (output value is in radians): [IO]	<code>awk '{print log(\$1+sqrt(\$1*\$1-1))}'</code>
show the hyperbolic arctangent of field1 (output value is in radians): [IO]	<code>awk '{print log((1+\$1)/(1-\$1))/2}'</code>
put the hyperbolic arctangent of \$a into variable 'x' (output value is in radians): [NN]	<code>x=\$(echo "scale=5;1((1+\$a)/(1-\$a))/2" bc -l)</code>
Prime factors	
show the prime factors of 5463: [NN]	<code>factor 5463</code>
show the prime factors of \$g: [NN]	<code>factor \$g</code>
show the prime factors of the numbers contained in <i>file1</i> : [NO]	<code>cat file1 factor</code>
list prime numbers contained in <i>file1</i> : [NO]	<code>cat file1 factor awk 'NF <= 2 {print \$2}'</code>
list non-prime numbers contained in <i>file1</i> along with their prime factors: [NO]	<code>cat file1 factor awk 'NF >= 2 {print \$0}'</code>

Desired Action	Command Line
show the prime factors of numbers from 1 to 1000: [NO]	<code>awk 'BEGIN {for (i=1;i<=1000;i++) print i}' factor</code>
show the prime factors of numbers from 1 to field1: [NO]	<code>awk '{for (i=1;i<=\$1;i++) print i}' factor</code>
show the prime factors of numbers from field1 to field2: [IO]	<code>awk '{for (i=\$1;i<=\$2;i++) print i}' factor</code>
show the prime factors of numbers from 1 to 'n': [NO]	<code>awk -v n=\$n 'BEGIN {for (i=1;i<=n;i++) print i}' factor</code>
show the prime factors of numbers from 'm' to 'n': [NO]	<code>awk -v m=\$m -v n=\$n 'BEGIN {for (i=m;i<=n;i++) print i}' factor</code>
Non-prime numbers	
show non-prime numbers and their prime factors from 1 to 1000: [NO]	<code>seq 1000 factor awk 'NF >= 2 {print \$0}'</code>
show non-prime numbers and their prime factors from 1 to 1000: [NO]	<code>awk 'BEGIN {for (i=1;i<=1000;i++) print i}' factor awk 'NF >= 2 {print \$0}'</code>
show non-prime numbers and their prime factors: [NO] Note: this lists without bound.	<code>awk 'BEGIN {for (i=1;i>0;i++) print i}' factor awk 'NF >= 2 {print \$0}'</code>
Prime numbers	
show prime numbers from 1 to 1000: [NO]	<code>seq 1000 factor awk 'NF <= 2 {print \$2}'</code>
show prime numbers from 1 to 1000: [NO]	<code>awk 'BEGIN {for (i=1;i<=1000;i++) print i}' factor awk 'NF <= 2 {print \$2}'</code>
show prime numbers: [NO] Note: this lists without bound.	<code>awk 'BEGIN {for (i=1;i>0;i++) print i}' factor awk 'NF <= 2 {print \$2}'</code>
show the first 10 prime numbers just above 10000: [NO]	<code>awk 'BEGIN {for (i=10000;i>1;i++) print i}' factor awk 'NF <= 2 {print \$2}' head -10</code>
show the prime numbers between 10000 and 12000: [NO]	<code>awk 'BEGIN {for (i=10000;i<=12000;i++) print i}' factor awk 'NF <= 2 {print \$2}'</code>
show the first 10 prime numbers just below 10000: [NO]	<code>awk 'BEGIN {for (i=10000;i>1;i--) print i}' factor awk 'NF <= 2 {print \$2}' head -10</code>
show prime twins from 1 to 1000: [NO]	<code>awk 'BEGIN {for (i=1;i<=1000;i++) print i}' factor awk 'NF <= 2 {print \$2}' awk 'NR<2{next} \$1==twin{print last" "twin} {last=\$1;twin=last+2;next}'</code>
show prime twins: [NO] Note: this lists without bound.	<code>awk 'BEGIN {for (i=1;i>0;i++) print i}' factor awk 'NF <= 2 {print \$2}' awk 'NR<2{next} \$1==twin{print last" "twin} {last=\$1;twin=last+2;next}'</code>

Desired Action	Command Line
show large primes (100 hex characters long) in hexadecimal: [NO] Note: this lists without bound.	cat /dev/urandom od -v -An -tx4 tr -d ' \t\n' fold -w 100 grep -v "[02468ace]\$" grep -v "^0" while read n;do openssl prime -hex \$n \ grep "is prime" cut -d' ' -f1;done
show large primes (100 hex characters long) in decimal: [NO] Note: this lists without bound.	cat /dev/urandom od -v -An -tx4 tr -d ' \t\n' fold -w 100 grep -v "[02468ace]\$" grep -v "^0" while read n;do openssl prime -hex \$n \ grep "is prime" cut -d' ' -f1 \ sed "s/^/ibase=16;/" bc;done
show large primes (200 hex characters long) in hexadecimal: [NO] Note: this lists without bound.	cat /dev/urandom od -v -An -tx4 tr -d ' \t\n' fold -w 200 grep -v "[02468ace]\$" grep -v "^0" while read n;do openssl prime -hex \$n \ grep "is prime" cut -d' ' -f1;done
show large primes (200 hex characters long) in decimal: [NO] Note: this lists without bound.	cat /dev/urandom od -v -An -tx4 tr -d ' \t\n' fold -w 200 grep -v "[02468ace]\$" grep -v "^0" while read n;do openssl prime -hex \$n \ grep "is prime" cut -d' ' -f1 \ sed "s/^/ibase=16;/" bc;done
Factorials	
show the factorial of field1: [IO]	awk 'BEGIN{f=1} {for(i=1;i<=\$1;i++) f+=f*i} END {print f}'
show the factorial of field2: [IO]	awk 'BEGIN{f=1} {for(i=1;i<=\$2;i++) f+=f*i} END {print f}'
show the factorial of the value in variable 'n': [NO]	awk -v n=\$n 'BEGIN {{f=1;for(i=1;i<=n;i++) f+=f*i} {print f}}'
Number sequences (arithmetic and geometric)	
generate the first 10 triangular numbers (one number per line): [NO]	awk 'BEGIN {for (i=1;i<=10;i++) {t=i+t; print t}}'
generate the first 10 triangular numbers (comma delimited): [NO]	awk 'BEGIN {for (i=1;i<=10;i++) {t=i+t; printf "%d,",t}}' sed 's/,,\$/\n/'
generate the first 25 triangular numbers (one number per line): [NO]	awk 'BEGIN {for (i=1;i<=25;i++) {t=i+t; print t}}'
generate the first field1 triangular numbers (one number per line): [IO]	awk '{for (i=1;i<=\$1;i++) {t=i+t; print t}}'
generate the first field1 triangular numbers (comma delimited): [IO]	awk '{for (i=1;i<=\$1;i++) {t=i+t; printf "%d,",t}}' sed 's/,,\$/\n/'
generate the first 10 Fibonacci numbers (one number per line): [NO]	awk 'BEGIN {t=0; b=1; for (i=1;i<=10;i++) {print t; p=t; t=t+b; b=p}}'
generate the first 35 Fibonacci numbers (one number per line): [NO]	awk 'BEGIN {t=0; b=1; for (i=1;i<=35;i++) {print t; p=t; t=t+b; b=p}}'

Desired Action	Command Line
generate the first field1 Fibonacci numbers (one number per line): [IO]	<pre>awk '{t=0; b=1; for (i=1;i<=\$1;i++) {print t; p=t; t=t+b; b=p}}'</pre>
generate the first 10 elements of an arithmetic progression starting at 4 with an increment of 6 (one number per line): [NO]	<pre>awk 'BEGIN {o=t=4; inc=6; for (i=1;i<=10;i++) {print t; t=o+(inc*i)}}'</pre>
generate the first field1 elements of an arithmetic progression starting at field2 with an increment of field3 (one number per line): [IO]	<pre>awk '{o=t=\$2; inc=\$3; for (i=1;i<=\$1;i++) {print t; t=o+(inc*i)}}'</pre>
generate the first 10 elements of a geometric progression starting at 4 with a common ratio of 6 (one number per line): [NO]	<pre>awk 'BEGIN {o=t=4; ratio=6; for (i=1;i<=10;i++) {print t; t=o*(ratio*i)}}'</pre>
generate the first field1 elements of a geometric progression starting at field2 with a common ratio of field3 (one number per line): [IO]	<pre>awk '{o=t=\$2; ratio=\$3; for (i=1;i<=\$1;i++) {print t; t=o*(ratio*i)}}'</pre>
Calculating network settings Note: see the man page for ipcalc limitations	Note: this is a calculator for subnet settings just as bc is a calculator for general math functions. The host (-h) option uses the local resolver to try to find a host name for that IP.
show all network settings for a NIC that would be set for 1.2.3.4/8: [NO]	<pre>ipcalc -bhmp 1.2.3.4/8</pre>
show all network settings for a NIC that would be set for 1.2.3.4 255.0.0.0: [NO]	<pre>ipcalc -bhmp 1.2.3.4 255.0.0.0</pre>
show broadcast address for a NIC that would be set for 1.2.3.4/8: [NO]	<pre>ipcalc -b 1.2.3.4/8</pre>
show host name for a NIC that would be set for 1.2.3.4/8: [NO]	<pre>ipcalc -h 1.2.3.4/8</pre>
show subnet mask for a NIC that would be set for 1.2.3.4/8: [NO]	<pre>ipcalc -m 1.2.3.4/8</pre>
show network for a NIC that would be set for 1.2.3.4/8: [NO]	<pre>ipcalc -n 1.2.3.4/8</pre>
show subnet prefix for a NIC that would be set for 1.2.3.4/8: [NO]	<pre>ipcalc -p 1.2.3.4/8</pre>
show subnet prefix for a NIC that would be set for 1.2.3.4 255.0.0.0: [NO]	<pre>ipcalc -p 1.2.3.4 255.0.0.0</pre>
Calculating times, dates, durations, calendars	
show the number of seconds to execute a command (bash internal): [NO]	<pre>time command</pre>
show the number of seconds to execute a command (external): [NO]	<pre>/usr/bin/time command</pre>

Desired Action	Command Line
show the number of seconds from January 1, 1970 00:00:00 to now: [NO]	date +%s
show the number of seconds from January 1, 1970 00:00:00 to some arbitrary timestamp: [NO]	date -d "hhmm CCYYMMDD" +%s Note: hh=hours, mm=minutes, CC=century, YY=year, MM=month, DD=day. Example: July 5, 2008 4:22pm would be "1622 20080705"
show the number of seconds from January 1, 1970 00:00:00 to some arbitrary day: [NO]	date -d "CCYYMMDD" +%s Note: CC=century, YY=year, MM=month, DD=day. Example: July 5, 2008 would be "20080705"
show the number of seconds from July 5, 2008 4:22pm to July 20, 2013 7:51pm: [NO]	echo "\$(date -d "1951 20130720" +%s) \ - \$(date -d "1622 20080705" +%s)" bc
show the number of minutes from July 5, 2008 4:22pm to July 20, 2013 7:51pm: [NO]	echo "scale=6; (\$(date -d "1951 \ 20130720" +%s) - \$(date -d "1622 \ 20080705" +%s))/60" bc
show the number of hours from July 5, 2008 4:22pm to July 20, 2013 7:51pm: [NO]	echo "scale=6; (\$(date -d "1951 \ 20130720" +%s) - \$(date -d "1622 \ 20080705" +%s))/3600" bc
show the number of days from July 5, 2008 4:22pm to July 20, 2013 7:51pm: [NO]	echo "scale=6; (\$(date -d "1951 \ 20130720" +%s) - \$(date -d "1622 \ 20080705" +%s))/86400" bc
show the number of weeks from July 5, 2008 4:22pm to July 20, 2013 7:51pm: [NO]	echo "scale=6; (\$(date -d "1951 \ 20130720" +%s) - \$(date -d "1622 \ 20080705" +%s))/604800" bc
show the number of the second of the minute (00-59): [NO]	date +%S
show the number of the minute of the hour (00-59): [NO]	date +%M
show the number of the hour of the day (00-23): [NO]	date +%H
show the number of the hour of the day (0-23): [NO]	date +%k
show the number of the day of the week (1-7=Mon-Sun): [NO]	date +%u
show the abbreviated name of the day of the week (Mon): [NO]	date +%a
show the abbreviated name of the day of the week of an arbitrary date: [NO]	date -d "CCYYMMDD" +%a Note: CC=century, YY=year, MM=month, DD=day. Example: July 5, 2008 would be "20080705"
show the full name of the day of the week (Monday): [NO]	date +%A
show the number of the day of the month (01-31): [NO]	date +%d
show the number of the day of the year (001-366): [NO]	date +%j

Desired Action	Command Line
show the number of the month of the year (01-12): [NO]	date +%m
show the abbreviated name of the day of the month (Jan): [NO]	date +%b
show the full name of the day of the month (January): [NO]	date +%B
show the year in yy format (00-99): [NO]	date +%y
show the century in cc format (00-99): [NO]	date +%C
show the year in yyyy format: [NO]	date +%Y
show AM or PM in uppercase: [NO]	date +%p
show am or pm in lowercase: [NO]	date +%P
show the current UTC time: [NO]	date -u
show the date in Month Day, Year format: [NO]	date "+%B %d, %Y"
show today's date in yyyy-mm-dd format: [NO]	date +%Y-%m-%d
show yesterday's date in yyyy-mm-dd format: [NO]	date -d "yesterday" +%Y-%m-%d
show tomorrow's date in yyyy-mm-dd format: [NO]	date -d "tomorrow" +%Y-%m-%d
show the time 30 seconds ago in hh:mm:ss format: [NO]	date -d "now -30 second" +%H:%M:%S
show the time 30 seconds from now in hh:mm:ss format: [NO]	date -d "now +30 second" +%H:%M:%S
show the time a minute ago in hh:mm:ss format: [NO]	date -d "now -1 minute" +%H:%M:%S
show the time a minute from now in hh:mm:ss format: [NO]	date -d "now +1 minute" +%H:%M:%S
show the time an hour ago in hh:mm format: [NO]	date -d "now -1 hour" +%H:%M
show the time an hour from now in hh:mm format: [NO]	date -d "now +1 hour" +%H:%M
show the time two hours ago in hh:mm format: [NO]	date -d "now -2 hour" +%H:%M
show the time two hours from now in hh:mm format: [NO]	date -d "now +2 hour" +%H:%M
show the time and date two days ago in yyyy-mm-dd-hh-mm format: [NO]	date -d "now -2 day" +%Y-%m-%d-%H-%M
show the date two days from now in yyyy-mm-dd format: [NO]	date -d "now +2 day" +%Y-%m-%d
show the time and date two weeks ago in yyyy-mm-dd-hh-mm format: [NO]	date -d "now -2 week" +%Y-%m-%d-%H-%M

Desired Action	Command Line
show the date two weeks from now in yyyy-mm-dd format: [NO]	<code>date -d "now +2 week" +%Y-%m-%d</code>
show the time and date two months ago in yyyy-mm-dd-hh-mm format: [NO]	<code>date -d "now -2 month" +%Y-%m-%d-%H-%M</code>
show the time and date two years ago in yyyy-mm-dd-hh-mm format: [NO]	<code>date -d "now -2 year" +%Y-%m-%d-%H-%M</code>
show the date for last Friday in yyyy-mm-dd format: [NO]	<code>date -d "last Friday" +%Y-%m-%d</code>
show the date for this coming Friday in yyyy-mm-dd format: [NO]	<code>date -d "this Friday" +%Y-%m-%d</code>
show the date for this coming Friday in yyyy-mm-dd format: [NO]	<code>date -d "next Friday" +%Y-%m-%d</code>
show the date a fortnight in the future in yyyy-mm-dd format: [NO]	<code>date -d "now + fortnight" +%Y-%m-%d</code>
show a calendar for this month, each week starting with the day specified by the locale: [NO]	<code>cal</code>
show a calendar for this month, each week starting with Sunday: [NO]	<code>cal -s</code>
show a calendar for this month, each week starting with Monday: [NO]	<code>cal -m</code>
show a calendar for last, this, and next month: [NO]	<code>cal -3</code>
show a calendar for this month, each day numbered from January 1: [NO]	<code>cal -j</code>
show a calendar for January 2008: [NO]	<code>cal 1 2008</code>
show a calendar for the year 2008: [NO]	<code>cal 2008</code>
show a calendar for the current year: [NO]	<code>cal -y</code>
show the date and time every second: [NO]	<code>while true; do date; sleep 1; done</code>

The Computational Versatility of Linux Shell

Before I researched the numeric computational abilities of certain shell commands, namely `awk`, `bc`, and `dc`, it did not seem like shell could do much in that area. Nothing could be farther from the truth, though. As this chapter demonstrates, a great variety of numeric calculations can be carried out from within a shell program. It was far more than what I could have previously imagined. Since that time, I have used those abilities freely, particularly in the calculating and reporting of system performance statistics.

It was also surprising to discover the wide range of data formatting and graphing choices. Not only can data be cleanly formatted in a text-based report, it can be displayed in multi-colored graphs of many types using standard image formats. Separate chapters in this book are dedicated to discussing the formatting and graphing of data.

Quite unexpected was the ability to generate arbitrarily long random hexadecimal numbers. Those could be fed into `openssl` to identify long prime numbers for cryptographic uses. For example, below is a routine that finds prime numbers that are 100 hexadecimal digits long.

```
cat /dev/urandom | od -v -An -tx4 | tr -d ' \t\n' | fold -w 100 |
grep -v "[02468ace]$" | grep -v "^0" | while read n ; do
openssl prime -hex $n|grep "is prime"|cut -d' ' -f1;done
```

Below is a version of the same that outputs in decimal digit form rather than hexadecimal form.

```
cat /dev/urandom | od -v -An -tx4 | tr -d ' \t\n' | fold -w 100 |
grep -v "[02468ace]$" | grep -v "^0" | while read n ; do
openssl prime -hex $n|grep "is prime"|cut -d' ' -f1|\
sed "s/^/ibase=16;/"|bc;done
```

The option on the `fold` command can be altered to vary the number of hex code digits that are sent to `openssl` for testing as a prime.

To verify operation, I generated a number of large hex numbers, fed them into `openssl`, then used the `factor` command to test the numbers that `openssl` had said were prime. Every number was verified by `factor` as prime.

4.20 Regular Expressions

What This Chapter Covers

- literals and metacharacters
- simple searches
- case-sensitive and case-insensitive searches
- negated searches, searching for the absence of something
- searches relative to the beginning or the end of the line
- searches involving characters that could be anything
- searches involving lists of characters
- searches involving ranges of characters
- searches involving POSIX character class definitions
- searches involving varying numbers of characters
- searches involving unprintable characters
- searches using the divide and conquer approach
- multi-part searches performed in parallel: the "OR" function
- multi-part searches performed in series: the "AND" function
- handy regular expressions

Literals and Metacharacters

The mathematician Stephen Kleene invented the theory behind what we now know as "regular expressions." The term "regular expressions" or "regexes" refers to statements constructed to search for particular patterns in character strings. Because of their foundation in mathematics and their mathematical completeness, they are incredibly versatile.

Regular expressions are composed of two kinds of characters:

1. literals -- characters that mean exactly what they appear to mean. For example, an "A" means "A".
2. metacharacters -- characters that have a meaning beside what they might appear to mean. For example, a dollar sign (\$) used as a metacharacter means, "end of line."

The backslash "\" is a metacharacter that removes the special meaning of any metacharacter that follows immediately behind it. Therefore, if we want to search for a dollar sign character rather than for the end of the line, we use "\\\$" to look for it. If we want to look for a backslash character, we use "\\\". An expression that uses a backslash to remove the special meaning from a metacharacter is sometimes referred to as an "escape sequence." The "Metacharacters: Characters with a Second Meaning" chapter in Section 4 explains each metacharacter in more detail.

We use regular expressions to find character strings of interest. The object of our search is called the "target string," and the expression we use in the search is called the "search expression." Ideally, we want to create a search expression that matches only our target string and ignores everything else. In some situations, that can pose a challenge. Here is an example. Suppose we have a simple log file composed of a single-column list of days of the week ("Monday," "Tuesday," etc.), and we want to find each reference to "Friday" or "Saturday". When we look for things in common with the spelling of those two days, we notice that each day has an "r" in it, so we might be tempted to set up a `grep` command like this:

```
grep "r"
```

That command is quite happy to find every "Friday" and "Saturday" for us...along with each "Thursday" because that day also contains an "r." Obviously, that poses a problem. Suppose we instead want to find only "Saturday" and "Sunday". So, we could set up a search expression like

```
grep "S"
```

and that works great for a while. Later on, though, we realize that sometimes we have to find "saturday" and "sunday" in all lower case. So we could alter the line to

```
grep -i "S" or grep "[Ss]"
```

in order to catch the days when they are all lower case. But either search expression could find more days than what we want because three other days of the week contain an "s" or an "S" if in all caps. Clearly, we have created another problem.

Unintentional matches often occur with regular expressions. Therefore, you must take thought not only for what you want to match, but also for what you want to ignore. Granted, the examples above represent trivial errors, but such errors happen, nevertheless.

Generally, the more thorough the search expression, the more likely you will match only what you intend to match. In other words, the most reliable way to find the desired target strings in this case would be to search for the entire name string, such as

```
grep -ie "saturday" -ie "sunday"
```

which ignores the case of all the characters, or

```
grep -e "[Ss]aturday" -e "[Ss]unday"
```

which ignores case only on the "s." The `[Ss]` alternative provides a more exact match. If you want an absolute exact match under these conditions, the search expression would be more like

```
grep -e "^[Ss]aturday$" -e "^[Ss]unday$"
```

which anchors the front and back of the names to the beginning and end of line. We will talk more about the use of anchors in search expressions shortly.

While more thorough search expressions tend to minimize false matches, a tradeoff exists there as well. If our search expression is too particular, it can miss valid matches, and the string we wanted to match will escape detection. Let's reconsider the search expression just above. What if a space or a tab character somehow sneaks in after "Saturday" or "Sunday" in the log file? What if the words sometimes appeared in all caps or all lower case? That sort of thing would cause the last two search expressions above to fail to match those log entries.

What is the message in this? Our search expression is a tradeoff: it must be narrow enough in focus not to match non-target strings, but it must be wide enough to match the target strings over the scope of all the forms those target strings might take. Too narrow, and the regex will not match enough, and it will allow strings that should match to escape. Too wide, and more strings will match than what should match. In the language of regular expressions, a good regex matches the target string every time and rejects non-target strings every time. The best way I know to determine the quality of a regex is to run a significant amount of real data through it and see if it behaves as expected. Real world data has a nasty habit of containing unexpected surprises. In my experience, the best regular expressions come from a combination of thoughtful consideration and real world testing with sizable samples of the actual data the regex will be

required to process. Only after we observe the regex perform properly with a large quantity of real data will our confidence in the regex be justified.

Although commands such as `awk`, `grep`, and `sed` are designed to work with regular expressions, you can encounter situations in which a particular regular expression will work fine with one command but not with other commands. For example, on my Linux desktop, the following regular expression works well to find numbers like "1-2-3" and "123-456-7890" when used with `awk`, but not with `grep` or with `sed`.

```
awk '/[0-9]+-[0-9]+-[0-9]+/'      # works
grep '[0-9]+-[0-9]+-[0-9]+'      # does not work
sed '/[0-9]+-[0-9]+-[0-9]+/!d'    # does not work
```

However, if you escape certain characters with a backslash, you can make those expressions work, as in

```
grep '[0-9]\+-[0-9]\+-[0-9]\+'   # works
sed '/[0-9]\+-[0-9]\+-[0-9]\+!/d' # works
```

You will need to use a backslash "\" to escape the following characters in `grep` and `sed` regular expressions:

- plus sign "+"
- question mark "?"
- pipe "|"
- left paren "("
- right paren ")"

The lesson: do not assume that every regular expression will work verbatim with every command in which you attempt to use it. As you can see above, sometimes you need to modify regexes when going from one command to another. That is why equivalent one-liners in this book are often listed for multiple commands. I like to show how a given regular expression is used with `awk`, `grep`, and `sed` to demonstrate the modifications that are needed for each of the various commands. Once you see how a family of related regexes can be used in all three of those commands, you can more easily grasp how they work. Once you understand how they work with each command, you can swap one command for another at will and can thereby use the best command for each situation. Many folks have never seen how `awk`, `grep`, and `sed` can be swapped for each other in certain situations as the following pages show. That knowledge is important because it opens extensive possibilities.

Although it is possible to embed unprintable characters such as various control characters in regular expressions, I recommend against it for maintainability reasons. If present in a script, they are easy to forget and do not appear on a printed listing. Commands such as `awk` and `sed` support printable versions of the unprintable characters you would most likely need to deal with such as `\n` (newline), `\t` (tab), and `\r` (return). Furthermore, `awk` supports octal and hexadecimal ASCII character designations in the forms `\nnn` (octal) and `\xnn` (hex). An embedded `printf` command could in theory enable nearly any other character manipulation command to work with unprintable characters. See the "Embedding Commands Within Commands" chapter in Section 4 for details.

Simple Searches

The following tables display three items: things we want to match, things we do *not* want to match, and search expressions to match only the target strings. Since `awk`, `grep`, and `sed` are the three commands most commonly used with regular expressions, examples for all three of those commands are provided for the simple searches below.

Examples of Simple Searches

Items to match	Items to not match	Regular expression
strings with a lowercase "p" like "ample" "pear" "apt"	strings lacking a lowercase "p" like "Pam" "lock" "velocity"	p as in awk '/p/' grep 'p' sed '/p/!d'
strings with an uppercase "J" like "Jay" "John" "Joey"	strings lacking an uppercase "J": "joy" "pot" "other"	J as in awk '/J/' grep 'J' sed '/J/!d'
strings with "am" like "ample" "Pam" "Sam"	strings lacking "am" like "AM" "aim" "velocity"	am as in awk '/am/' grep 'am' sed '/am/!d'
strings with "oc" like "lock" "octopus" "velocity"	strings lacking "oc" like "October" "fire" "spear"	oc as in awk '/oc/' grep 'oc' sed '/oc/!d'
strings with "1/2" like "1/23" "71/25" "301/204"	strings lacking "1/2" like "nothing" "1/4" "3/2"	1\2 as in awk '/1\2/' grep '1\2' sed '/1\2/!d'
strings with "5.3" like "15.3" "35.3" "95.36"	strings lacking "5.3" like "5.2" "23.5" "85.7"	5\.3 as in awk '/5\.3/' grep '5\.3' sed '/5\.3/!d'
strings with "a[3" like "aqua[3" "terra[3" "ha[3"	strings lacking "a[3" like "a3[" "bat" "b[4"	a\[3 as in awk '/a\[3/' grep 'a\[3' sed '/a\[3/!d'
strings with "r^2" like "A=pr^2" "rear^2" "far^2"	strings lacking "r^2" like "r^3" "a^2" "r*2"	r\^2 as in awk '/r\^2/' grep 'r\^2' sed '/r\^2/!d'
strings with "\$v" like "\$var" "\$v2" "\$virgil"	strings lacking "\$v" like "v\$" "\$VAR" "\$g"	\\$v as in awk '/\\$v/' grep '\\$v' sed '/\\$v/!d'

As you can see above, `awk`, `grep`, and `sed` can be used interchangeably in many string-matching situations. Depending on the action to be taken on the matched string, one of those three commands will typically be better to use than the other two.

Case-sensitive and Case-insensitive Searches

Sometimes we want to match uppercase or lowercase along with the actual letters, and sometimes we do not care about case. Those are referred to as "case-sensitive" and case-insensitive" searches. They fall into three general categories:

1. Match of letter and case for every letter
2. Match of letter and case for only certain letters
3. Match of letters only without regard to case

As in the table above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Case-sensitive and Case-insensitive Searches

Items to match	Items to not match	Regular expression
strings containing exact case matches to "Rob" like "Rob" "Robert" "Robbie"	strings lacking "Rob" like "rob" or "robber" "Janet"	Rob as in awk '/Rob/' grep 'Rob' sed '/Rob/!d'
strings containing first letter upper or lower case matches to "Rob" or "rob" like "Robbie" "robber"	strings not containing "Rob" or "rob" like "bob" "job" "crank"	[Rr]ob as in awk '/[Rr]ob/' grep '[Rr]ob' sed '/[Rr]ob/!d'
strings containing case-insensitive matches to "rob" like "ROB" "rOb" "RoB" "roBber"	strings not containing the letters "rob" in any form like "bob" "job" "crank"	[Rr][Oo][Bb] or rob or ROB as in awk 'tolower(\$0)~/rob/' awk 'toupper(\$0)~/ROB/' awk '/[Rr][Oo][Bb]/' grep -i 'rob' grep '[Rr][Oo][Bb]' sed '/[Rr][Oo][Bb]/!d'

Negated Searches, Searching for the Absence of Something

Matching something present in a string is useful, but so is the ability to match something not present there. The caret "^" gives us the ability to say "do not match the items that follow" within a set of square brackets.

The table below shows how to match on the absence of various characters. Note that when looking for the absence of a letter as in `un[^t]` there still must be some letter present in place of the undesired "t". In other words, "sun" alone will not match `un[^t]` because no non-t character occurs in that word to match the `^t` specification. On the other hand, if a space occurred after the word "sun", the `un[^t]` specification would match because it would see the space as a non-t character.

As in the table above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Negated Searches

Items to match	Items to not match	Regular expression
strings containing "ver" but not "vers" like "lever" "very" "vertigo"	strings containing "vers" or lacking "ver" like "verse" "version" "berry"	<pre>ver[[^]s] as in awk '/ver[[^]s]/' grep 'ver[[^]s]'</pre> <pre>sed '/ver[[^]s]/!d'</pre> <p>Note: the "[[^]s]" spec must see a non-s character after the "ver" in order to match.</p>
strings containing "un" but not "unt" like "unsold" "undo" "under"	strings containing "unt" or lacking "un" like "untold" "until" "cranky" "sun"	<pre>un[[^]t] as in awk '/un[[^]t]/' grep 'un[[^]t]'</pre> <pre>sed '/un[[^]t]/!d'</pre> <p>Note: the "[[^]t]" spec must see a non-t character after the "un" in order to match.</p>
strings ending with "day" but not "sday" like "Monday" "Friday" "Saturday" "Sunday"	strings ending with "sday" or not ending with "day" like "Tuesday" "Wednesday" "Thursday"	<pre>[[^]s]day\$ as in awk '/[[^]s]day\$/' grep '[[^]s]day\$'</pre> <pre>sed '/[[^]s]day\$/!d'</pre> <p>Note: the "[[^]s]" spec must see a non-s character before the "day" in order to match.</p>
strings ending with "day" but not "nday" or "sday" like "Friday" "Saturday" "today"	strings ending with "nday" or "sday" or not ending with "day" like "Monday" "Tuesday" "Wednesday" "Thursday" "Sunday" "other"	<pre>[[^]ns]day\$ as in awk '/[[^]ns]day\$/' grep '[[^]ns]day\$'</pre> <pre>sed '/[[^]ns]day\$/!d'</pre> <p>Note: the "[[^]ns]" spec must see a non-n, non-s character before the "day" in order to match.</p>
strings ending with "day" but not "nday" or "rday" or "sday" like "Friday" "today" "someday"	strings ending with "nday" or "rday" or "sday" or not ending with "day" like "Monday" "Tuesday" "Wednesday" "Thursday" "Saturday" "Sunday" "other"	<pre>[[^]nrs]day\$ as in awk '/[[^]nrs]day\$/' grep '[[^]nrs]day\$'</pre> <pre>sed '/[[^]nrs]day\$/!d'</pre> <p>Note: the "[[^]nrs]" spec must see a non-n, non-r, non-s character before the "day" in order to match.</p>
strings ending with "day" but not "iday" or "nday" or "rday" or "sday" like "today" "someday"	strings ending with "iday" or "nday" or "rday" or "sday" or not ending with "day" like "Monday" "Tuesday" "Wednesday" "Thursday" "Friday" "Saturday" "Sunday"	<pre>[[^]inrs]day\$ as in awk '/[[^]inrs]day\$/' grep '[[^]inrs]day\$'</pre> <pre>sed '/[[^]inrs]day\$/!d'</pre> <p>Note: the "[[^]inrs]" spec must see a non-i, non-n, non-r, non-s character before the "day" in order to match.</p>
strings containing "b" <something not "o" and not "z"> "t" like "bat" "bet" "bit"	strings containing "bot" or "bzt" or something else like "BAT" "bta" "never"	<pre>b[[^]oz]t as in awk '/b[[^]oz]t/' grep 'b[[^]oz]t'</pre> <pre>sed '/b[[^]oz]t/!d'</pre>

Items to match	Items to not match	Regular expression
strings containing "ent" but not at the beginning like "rent" "sent" "spent" "denting"	strings having "ent" at the beginning or not containing "ent" at all like "enter" "entry" "march"	[^^]ent as in awk '/[^^^]ent/' grep '[^^]ent' sed '/[^^^]ent/!d' Note: the "[^^]" spec must see a character before the "ent" in order to match.
strings containing "ent" but not at the end of the line like "enter" "entry" "denting"	strings having "ent" at the end or not containing "ent" at all like "rent" "sent" "march"	ent[^\$] as in awk '/ent[^\$]/' grep 'ent[^\$]' sed '/ent[^\$]/!d' Note: the "[^\$]" spec must see a character after the "ent" in order to match.

Searches Relative to the Beginning or the End of the Line

Sometimes we want to match characters at the beginning or the end of a line. The technique for finding string patterns relative to the beginning or end of the line is called "anchoring." The anchor metacharacters are "^" for the beginning of the line and "\$" for the end of the line. For example, if we want to find "the" at the beginning of the line, we would search for "^the". If we wanted to find "forever" at the end of the line, we would search for "forever\$".

As in the tables above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Searches Relative to the Beginning or End of the Line

Items to match	Items to not match	Regular expression
empty lines, consecutive newlines	lines with any character before the newline	^\$ as in awk '/^\$/' grep '^\$' sed '/^\$/!d'
lines composed of nothing but one space	any line composed of something other than one space like "I" "at" "before"	^ \$ as in awk '/^ \$/' grep '^ \$' sed '/^ \$/!d'
lines composed of nothing but "3"	any line composed of something other than "3" like "33" "587" "be"	^3\$ as in awk '/^3\$/' grep '^3\$' sed '/^3\$/!d'
lines composed of nothing but "this"	any line composed of something other than just "this" like "this way" "this hope" "other stuff"	^this\$ as in awk '/^this\$/' grep '^this\$' sed '/^this\$/!d'
strings with "b" at the beginning of the line like "boy" "barter" "blue"	strings lacking a "b" at the beginning of the line like "Pam" "lock" "velocity"	^b as in awk '/^b/' grep '^b' sed '/^b/!d'

Items to match	Items to not match	Regular expression
strings with "e" at the end of the line like "here" "cleave" "be"	strings lacking an "e" at the end of the line like "after" "broken" "plain"	e\$ as in awk '/e\$/' grep 'e\$' sed '/e\$!/d'
strings with "un" at the beginning of the line like "unto" "under" "unable"	strings lacking a "un" at the beginning of the line like "and" "fun" "utensil"	^un as in awk '/^un/' grep '^un' sed '/^un!/d'
strings with "able" at the end of the line like "able" "table" "cable"	strings lacking an "able" at the end of the line like "title" "label" "plain"	able\$ as in awk '/able\$/' grep 'able\$' sed '/able\$!/d'
strings with "ten" at the beginning of the line like "ten" "tent" "tenacious"	strings lacking a "ten" at the beginning of the line like "ton" "men" "intent"	^ten as in awk '/^ten/' grep '^ten' sed '/^ten!/d'

Searches Involving Characters that Could be Anything

The period "." acts as a placeholder for one character. It can be used in the place of any character you do not know or do not care about in your search expression. For example, if you want to search for a string that contains "an" but has at least one unknown character in front of it and behind it, the search expression would be ".an.". That would match words like "band," "sand," and "sandal." To remove the special meaning of the period, use the backslash as in "\.an\..".

As in the tables above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Searches Involving Characters that Could be Anything

Items to match	Items to not match	Regular expression
strings containing at least one initial character then "o" like "do" "ado" "go" "not"	strings lacking a character followed by "o"	.o as in awk '/.o/' grep '.o' sed '/.o!/d'
strings containing "c" followed by exactly two characters, then "t" like "cent" "scent" "cart"	strings lacking "c" <something><something> "t" like "cat" "carot" "gun"	c..t as in awk '/c..t/' grep 'c..t' sed '/c..t!/d'
strings containing "art" plus at least one character preceding it like "Bart" "cart" "part" "apart"	strings lacking "art" or at least one character before it like "artistic" "aft" "draft"	.art as in awk '/.art/' grep '.art' sed '/.art!/d'
strings containing "h" <some character> "rd" like "hard" "herd"	strings lacking that pattern like "horse" "bullet" "gun"	h.rd as in awk '/h.rd/' grep 'h.rd' sed '/h.rd!/d'

Items to match	Items to not match	Regular expression
strings containing "har" followed by at least one character like "hark" "harm" "harp" "hart"	strings lacking that pattern like "attack" "Buford" "fence"	har. as in awk '/har./' grep 'har.' sed '/har./!d'
strings containing at least one character followed by "ar" like "bar" "cart" "far" "gar" "jar" "parse" "stars"	strings lacking that pattern like "run" "can" "hurry" "twist" "argue"	.ar as in awk '/.ar/' grep '.ar' sed '/.ar/!d'
strings containing "ba" plus at least one more character like "bat" "bar" "baggage"	strings lacking "ba" plus at least one more character like "ba" "dog" "cat"	ba. as in awk '/ba./' grep 'ba.' sed '/ba./!d'

Searches Involving Lists of Characters

A placeholder like the period can solve many search problems, but sometimes we have to narrow things down more precisely than just some character in a particular spot. That is when lists become handy.

Lists of possible character matches are enclosed in square brackets "[...]". For instance, the expression `w[aioun]` would match "wan," "win," and "won." As you can see from this example, one and only one character out of the bracketed set needs to or can match. In other words, `s[aei]t` would match "sat," "set," and "sit," but not "seat" even though both "e" and "a" appear in the list.

If you want to look for possible misspellings or alternate spellings of a word, character lists can be useful. The expression `gr[ae]y` matches both "gray" and "grey." The expression `sep[ae]rate` matches "separate" and "seperate." The expression `[CK]rist[ei]n` matches "Cristen," "Kristen," "Cristin," and "Kristin."

As in the tables above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Searches Involving Lists of Characters

Items to match	Items to not match	Regular expression
strings containing "do" "go"	strings lacking "do" and "go" like "Do" "get" "Lee"	[dg]o as in awk '/[dg]o/' grep '[dg]o' sed '/[[dg]o/!d'
strings containing "hard" "herd"	strings lacking "hard" and "herd" like "horse" "bullet" "gun"	h[ae]rd as in awk '/h[ae]rd/' grep 'h[ae]rd' sed '/h[ae]rd/!d'
strings containing "hard" "hark" "harm" "harp" "hart"	strings lacking those words like "attack" "Buford" "fence"	har[dkmpt] as in awk '/har[dkmpt]/' grep 'har[dkmpt]' sed '/har[dkmpt]/!d'

Items to match	Items to not match	Regular expression
strings containing "bar" "car" "far" "gar" "jar" "par"	strings lacking those words like "fat" "can" "hurry"	[bcfgjp]ar as in awk '/[bcfgjp]ar/' grep '[bcfgjp]ar' sed '/[bcfgjp]ar/!d'
strings containing "bar" "car" "bat" "cat"	strings lacking those words like "bet" "can" "hurry"	[bc]a[rt] as in awk '/[bc]a[rt]/' grep '[bc]a[rt]' sed '/[bc]a[rt]/!d'
strings containing "Bart" "bart" "cart"	strings lacking those words like "Bert" "cot" "hurry"	[Bbc]art as in awk '/[Bbc]art/' grep '[Bbc]art' sed '/[Bbc]art/!d'

Searches Involving Ranges of Characters

As is plainly seen in the examples above, lists of characters can be useful in regular expressions. When a number of contiguous characters is involved in a list, that list can be described as a range. For example, the list [abcde] can be described as the range [a-e]. Digits can be described as the range [0-9]. The lowercase alphabet is [a-z] and the uppercase, [A-Z].

Multiple ranges can be used together within a single set of brackets. The entire alphabet can be stated as [A-Za-z]. Hexadecimal numbers are [0-9A-Fa-f]. The order of the ranges within the brackets does not matter, so hexadecimals could be stated just as validly by [A-F0-9a-f] or by [a-fA-F0-9].

As in the tables above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Searches Involving Ranges of Characters

Items to match	Items to not match	Regular expression
strings containing any lowercase letter like "Joe" "fire" "win"	strings lacking lowercase letters like "HIT" "23454" "!@#%\$"	[a-z] as in awk '/[a-z]/' grep '[a-z]' sed '/[a-z]/!d'
strings containing any uppercase letter like "Joe" "HIT" "Win!"	strings lacking uppercase letters like "shoot" "23454" "!@#%\$"	[A-Z] as in awk '/[A-Z]/' grep '[A-Z]' sed '/[A-Z]/!d'
strings containing any digit like "17" "76" "313" "23454"	strings lacking digits like "shoot" "baboon" "!@#%\$"	[0-9] as in awk '/[0-9]/' grep '[0-9]' sed '/[0-9]/!d'
strings containing any hexadecimal digit like "1A" "ED" "24" "FCFF"	strings lacking hexadecimal digits like "shoot" "town" "!@#%\$"	[0-9A-Fa-f] as in awk '/[0-9A-Fa-f]/' grep '[0-9A-Fa-f]' sed '/[0-9A-Fa-f]/!d'
strings containing "a" or "b" or "c" or "x" or "y" or "z" like "another" "bike" "many" "jazz"	strings lacking those letters like "hit" "miss" "shoot"	[a-cx-z] as in awk '/[a-cx-z]/' grep '[a-cx-z]' sed '/[a-cx-z]/!d'

Items to match	Items to not match	Regular expression
strings containing any digit or uppercase letter like "Joe" "HIT" "Win!" "phase1" "23454"	strings lacking those characters like "shoot" "!@#\$\$%" "bullet"	[0-9A-Z] as in awk '/[0-9A-Z]/' grep '[0-9A-Z]' sed '/[0-9A-Z]/!d'
strings containing the digits "5" through "9" or the letters "f" through "m" like "may1" "550" "fan"	strings lacking those characters like "shoot" "!@#\$\$%" "ban"	[5-9f-m] as in awk '/[5-9f-m]/' grep '[5-9f-m]' sed '/[5-9f-m]/!d'

Searches Involving POSIX Character Class Definitions

In Section 1, the chapter "POSIX Character Class Definitions" shows which classes exist and which characters belong to each class. Below is a brief summary of the classes with regular expressions equivalent to them in ASCII.

- [:alnum:] -- alphanumeric characters -- [0-9A-Za-z]
- [:alpha:] -- alphabetic characters -- [A-Za-z]
- [:blank:] -- horizontal whitespace -- [\011\040]
- [:cntrl:] -- control characters -- [\000-037\177]
- [:digit:] -- decimal digits -- [0-9]
- [:graph:] -- printable characters without <space> -- [\041-\176]
- [:lower:] -- lowercase letters -- [a-z]
- [:print:] -- printable characters including <space> -- [\040-\176]
- [:punct:] -- punctuation -- [\041-\057\072-\100\133-\140\173-\176]
- [:space:] -- horizontal and vertical whitespace -- [\011-\015\040]
- [:upper:] -- uppercase letters -- [A-Z]
- [:xdigit:] -- hexadecimal digits -- [0-9A-Za-z]

As you can see, the class names above apply to specific ranges or collections of characters. And it does indeed make sense to apply names to them. It is much more convenient to specify the printable characters by the name "[[:print:]]" than to have to list each character in that class every time you want to match them. Even in range notation it would be cumbersome.

When using class names with `awk`, `grep`, and `sed`, you must enclose the class name in another set of square brackets, as in `[[[:alnum:]]]`. A `grep` command that looks for lines containing alphanumerics would then be

```
grep '[[[:alnum:]]]'
```

When using class names with the `tr` command, though, the names do not require the second set of square brackets. For example, a filter to delete everything but alphanumeric characters and newlines would look like this:

```
tr -dc '[:alnum:]\n'
```

Note: the `grep` and `sed` commands do not accept octal character designations such as `"\011"`, therefore the one-liners in the table below reflect that fact. Although `awk` can accept certain octal values, those values

must decode to valid search characters, so it is generally best practice to use alphanumeric search characters with `awk` when possible.

As in the tables above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Searches Involving POSIX Character Class Definitions

Items to match	Items to not match	Regular expression
strings containing alphanumeric characters like "great" "day" "marching" "123" "732"	strings lacking digits and alphabetic characters like "#\$%^" "&*(@"	<code>[0-9A-Za-z]</code> or <code>[[[:alnum:]]]</code> as in <code>awk '/[0-9A-Za-z]/'</code> <code>awk '/[[[:alnum:]]]/'</code> <code>grep '[0-9A-Za-z]'</code> <code>grep '[[[:alnum:]]]'</code> <code>sed '/[0-9A-Za-z]/!d'</code> <code>sed '/[[[:alnum:]]]/!d'</code>
strings containing alphabetic characters like "that" "good" "news"	strings lacking alphabetic characters like "4253" ")(*&"	<code>[A-Za-z]</code> or <code>[[[:alpha:]]]</code> as in <code>awk '/[A-Za-z]/'</code> <code>awk '/[[[:alpha:]]]/'</code> <code>grep '[A-Za-z]'</code> <code>grep '[[[:alpha:]]]'</code> <code>sed '/[A-Za-z]/!d'</code> <code>sed '/[[[:alpha:]]]/!d'</code>
strings containing certain "blank" characters, namely, <horizontal tab> and <space>	strings lacking horizontal tabs and spaces	<code>[\011\040]</code> or <code>[[[:blank:]]]</code> as in <code>awk '/[\011\040]/'</code> <code>awk '/[[[:blank:]]]/'</code> <code>grep '[[[:blank:]]]'</code> <code>sed '/[[[:blank:]]]/!d'</code>
strings containing control characters like <htab> <vtab>	strings lacking control characters like "a" "quick" "brown" "fox"	<code>[[[:cntrl:]]]</code> as in <code>awk '/[[[:cntrl:]]]/'</code> <code>grep '[[[:cntrl:]]]'</code> <code>sed '/[[[:cntrl:]]]/!d'</code>
strings containing digits like "7453" "4539"	strings lacking digits like "happy" "days"	<code>[0-9]</code> or <code>[[[:digit:]]]</code> as in <code>awk '/[0-9]/'</code> <code>awk '/[[[:digit:]]]/'</code> <code>grep '[0-9]'</code> <code>grep '[[[:digit:]]]'</code> <code>sed '/[0-9]/!d'</code> <code>sed '/[[[:digit:]]]/!d'</code>
strings containing printable characters not including <space> like "first" "2nd" "3rd!"	strings lacking printable characters like <htab> <vtab>	<code>[[[:graph:]]]</code> as in <code>awk '/[[[:graph:]]]/'</code> <code>grep '[[[:graph:]]]'</code> <code>sed '/[[[:graph:]]]/!d'</code>

Items to match	Items to not match	Regular expression
strings containing lowercase characters "Head" "shoulders" "123contact" "jump!"	strings lacking lowercase characters like "BUD" "65345"	[a-z] or [[:lower:]] as in awk '/[a-z]/' awk '/[[:lower:]]/' grep '[a-z]' grep '[[:lower:]]' sed '/[a-z]/!d' sed '/[[:lower:]]/!d'
strings containing printable characters including <space> like "that way" "hi?" "34u"	strings lacking printable characters <null> <backspace>	[[:print:]] as in awk '/[[:print:]]/' grep '[[:print:]]' sed '/[[:print:]]/!d'
strings containing punctuation characters like "1,2,3" "5.9" "no!"	strings lacking punctuation characters "baby" "face" "Mollie"	[[:punct:]] as in awk '/[[:punct:]]/' grep '[[:punct:]]' sed '/[[:punct:]]/!d'
strings containing horizontal and vertical whitespace characters like <htab> <vtab> <newline>	strings lacking horizontal and vertical whitespace characters "no" "whitespace" "here"	[[:space:]] as in awk '/[[:space:]]/' grep '[[:space:]]' sed '/[[:space:]]/!d'
strings containing uppercase characters "Jerry" "St. Lo" "Big"	strings lacking uppercase characters "boy" "dog" "farm"	[A-Z] or [[:upper:]] as in awk '/[A-Z]/' awk '/[[:upper:]]/' grep '[A-Z]' grep '[[:upper:]]' sed '/[A-Z]/!d' sed '/[[:upper:]]/!d'
strings containing hexadecimal digits like "AF" "90DF" "123"	strings lacking hexadecimal digits "trick" "guppy" "jimmy"	[0-9A-Fa-f] or [[:xdigit:]] as in awk '/[0-9A-Fa-f]/' awk '/[[:xdigit:]]/' grep '[0-9A-Fa-f]' grep '[[:xdigit:]]' sed '/[0-9A-Fa-f]/!d' sed '/[[:xdigit:]]/!d'

Searches Involving Varying Numbers of Characters

Up to this point in this chapter, we have examined only what some might call "well-behaved" target strings. The target strings have had a set length and a set content. Not all target strings are like that, though.

Sometimes we want to look for strings like "Dad" or "Daddy," but not "Dadd." In other words, the target string could be one length or another but not something in between. Or the target string could be a number of lengths with a number of different contents. These are not well-behaved target strings. That being the case, we have a number of metacharacters to use.

- The parentheses "(...)" define a group of characters as in the expression `(un)?do`, that would match "do" and "undo". Because of the parentheses, the "un" letters are considered one entity, and that entity in this example can either be there or not there, but not half-there. It's all or nothing.
- The square brackets "[...]" define a list of characters as in the expression `b[ae]d`, that would match "bad" and "bed." Only one of those characters in that list can take the place of the middle character at a time to match the words "bad" or "bed." We have already seen many examples of the square brackets in use.
- The period "." specifies a 1-character match of any character other than newline. For example, the expression `for. .` would match "forks" and "forth".
- The question mark "?" specifies a 0- or 1-time match of the character just before it. For example, the expression `FAA?` allows the second "A" to be present or absent in the target string. That search expression would therefore match both "FA" and "FAA". Groups of characters can be handled in this way, too. The expression `exert(ion)?` would match either "exert" or "exertion".
- The asterisk "*" specifies a 0 to an unlimited number of matches of the character just before it. For example, the expression `5543*` allows "3" to occur an unlimited number of times. Or it could be absent and still be considered a match. Groups are also supported. The expression `M(iss)*ippi` would match "Mississippi", but it would also match "Mippi" and "Missississississippi".
- The plus sign "+" specifies a 1 to unlimited number of matches of the character just before it. For example, the expression `5543+` allows "3" to occur an unlimited number of times. But it must appear at least once in order to be considered a match. Another way to match at least one occurrence would be `55433*` in which the first "3" must be present, but the second or any number of other "3"s right after that need not be. Groups are also supported. The expression `M(iss)+ippi` would match "Mississippi", but it would also match "Missippi" and "Missississississippi". The asterisk method to match all of those would be `M(iss)(iss)*ippi`
- The vertical bar "|" specifies an either/or condition: either one part of the expression can match, or the other part can. For example, the expression `(r|(sc))ent` would match both "rent" and "scent". If we wanted to get fancy, the regular expression `(r|s|(sc)|(asc)|(pres)|(cresc))ent` would match "rent", "sent", "scent", "ascent", "present", or "crescent".

With the above arsenal of metacharacters along with the ones we examined previously, it might be difficult to imagine a target string for which a workable search expression could not be crafted.

As you may recall, certain metacharacters used with the `grep` and `sed` commands need to be escaped with a backslash "\" in order to behave the same as with `awk`. Those metacharacters are plus sign "+", question mark "?", pipe "|", left paren "(", and right paren ")".

As in the tables above, examples in each case are provided for `awk`, `grep`, and `sed`.

Examples of Searches Involving Variable Numbers of Characters

Items to match	Items to not match	Regular expression
strings containing "g" followed by "e" like "age" "gone" "gorge" "gauge" "gauges"	strings like "ego" "gantry" "Great"	g.*e as in awk '/g.*e/' grep 'g.*e' sed '/g.*e/!d'
strings containing "h" followed by "e" followed by "s" like "heats" "hews" "shrews" "sheets"	strings like "she" "hear" "west" "hats"	h.*e.*s as in awk '/h.*e.*s/' grep 'h.*e.*s' sed '/h.*e.*s/!d'
strings containing "had" "head" "hard" "heard"	strings like "heed" "hear" "hearth"	he?ar?d as in awk '/he?ar?d/' grep 'he\?ar\?d' sed '/he\?ar\?d/!d'
strings containing "Dad" "Daddy"	strings like "dad" "Dadd" "Daddi"	Dad(dy)? as in awk '/Dad(dy)?/' grep 'Dad\ (dy)\ \?' sed '/Dad\ (dy)\ \?/!d'
strings containing "lad" "ladder"	strings like "dal" "ladd" "laddar"	lad(der)? as in awk '/lad(der)?/' grep 'lad\ (der)\ \?' sed '/lad\ (der)\ \?/!d'
strings containing "let" "letter"	strings like "lot" "ladder"	let(ter)? as in awk '/let(ter)?/' grep 'let\ (ter)\ \?' sed '/let\ (ter)\ \?/!d'
strings containing "excite" "excitement"	strings like "excitation" "exit" "cement"	excite(ment)? as in awk '/excite(ment)?/' grep 'excite\ (ment)\ \?/' sed '/excite\ (ment)\ \?/!d'
strings containing "decide" "undecide" "decided" "undecided"	strings like "deciding" "decision" "decade"	(un)?decided? as in awk '/(un)?decided?/' grep '\ (un)\ \?decided\?' sed '/\ (un)\ \?decided\?/!d'
strings containing "red" "reed"	strings like "Red" "rod" "rot" "reef"	re+d or ree?d or ree*d as in awk '/re+d/' awk '/ree?d/' awk '/ree*d/' grep 're\+d' grep 'ree\?d' grep 'ree*d' sed '/ree\+d/!d' sed '/ree\?d/!d' sed '/ree*d/!d'

Items to match	Items to not match	Regular expression
strings containing "cares" "caress"	strings like "cart" "carres" "prison"	cares+ or caress? or caress* as in awk '/ cares+/' awk '/ caress?/' awk '/ caress*/' grep ' cares\+' grep ' caress\?' grep ' caress*' sed '/ cares\+!/d' sed '/ caress\?!/d' sed '/ caress*/!d'
strings containing "space" "spade" "spare" "spate"	strings like "stale" "glade" "stare"	spa(c d r t)e or spa[cdrt]e as in awk '/ spa(c d r t)e/' awk '/spa[cdrt]e/' grep 'spa[cdrt]e' sed '/spa[cdrt]e!/d'
strings containing "work" "worth"	strings like "wart" "worse"	wor(k (th)) as in awk '/wor(k (th))/' grep 'wor\(k \(th\)\)' sed '/wor\(k \(th\)\)/!d'
strings containing "doubt" "double"	strings like "dubs" "buds"	doub(t (le)) as in awk '/doub(t (le))/' grep 'doub\(t \(le\)\)' sed '/doub\(t \(le\)\)/!d'
strings containing "mat" "what"	strings like "met" "whet"	(m (wh))at as in awk '/(m (wh))at/' grep '\(m \(wh\)\)at' sed '/\(m \(wh\)\)at!/d'
strings containing "bead" "bears"	strings like "beer" "bear"	bea(d (rs)) as in awk '/bea(d (rs))/' grep 'bea\(d \(rs\)\)' sed '/bea\(d \(rs\)\)/!d'
strings containing "tend" "tenth"	strings like "tens" "ton" "turtle"	ten(d (th)) as in awk '/ten(d (th))/' grep 'ten\(d \(th\)\)' sed '/ten\(d \(th\)\)/!d'

Searches Involving Unprintable Characters

Let's take on what might at first glance appear to be an impossible challenge: matching unprintable characters such as <tab> or <alert> or 003 octal. The first problem is how to tell the script to look for such a thing. First, does any our three main search commands `awk`, `grep`, or `sed` have any built-in means to signify unprintable characters? Yes, `awk` can cover the entire range 000-377 octal. Below is an example that matches octal 3

```
awk '/\003/{print $0}'
```

The `grep` command is not equipped to deal with such characters. But it does permit us to cheat a bit by allowing another command to generate the character(s) for which `grep` will search. In other words, `grep "\003"` will not work, but we can match it with

```
grep "$(printf "\003") "          # or
grep "$(printf "\\003") "
```

I have not found `grep` able to match a null (octal 000) using this method, but it is able to match certain other unprintables. In my testing, it could match some of but not the whole range 001-377.

Newer versions of the `sed` command are able to match unprintables and output unprintables using this method. The line below matches octal 3 and changes it to bell (octal 7).

```
sed "s/$(printf "\003")/$(printf "\007")/g"
```

In my testing, the line below worked as well.

```
sed "s/$(printf "\\003")/$(printf "\\007")/g"
```

Some older versions of `sed` do not lend themselves to this kind of matching. Some can, however, match certain unprintables based on their backslash designations such as `"\b"` or `"\t"`.

Searches Using the Divide and Conquer Approach

Despite the versatility of the metacharacters and search techniques we have seen, there is always the "X-factor," the situation you cannot anticipate or design for based on what you already know. No matter what you do or how you plan, it can emerge from nowhere and bite you in the back pockets before you know what is happening. It is a special case.

Special cases require special techniques. Consider this the "special weapons and tactics" part of the chapter.

The math that was developed by Stephen Kleene indicates that any complex target string can be broken down into simpler parts. Broken into enough parts, each part of the target string can be coded for and handled appropriately. But how can a search be broken into multiple parts?

Multi-part Searches Performed in Parallel: the "OR" function

The `awk` command is actually a programming language. As such, it possesses many abilities that often go unused. Breaking down a search into multiple parts with `awk` is easy to do. See the example below.

```
awk '
/good/ {print $0;next}
/bad/  {print $0;next}
/ugly/ {print $0;next}
(next}'
```

Above we see a search executed in three parallel parts. If `awk` sees the word "good" or "bad" or "ugly" in a string, it outputs the associated line. In essence, we have three separate unrelated searches being performed at the same time on the same input data stream. The `next` is there to prevent the same line from being output multiple times if it contains more than one of the search strings. A similar thing can be done with `grep` and `sed` as is shown below.

```
grep -e "good" -e "bad" -e "ugly"
sed '/\(good\)\\|\(bad\)\\|\(ugly\)\/!d'
```

The three examples above are functionally equivalent. They perform an "OR" function in applying conditions to the strings to see which ones match. Such a configuration with any of these three commands enables us to execute in parallel any collection of searches on a stream of data.

Multi-part Searches Performed in Series: the "AND" function

Perhaps we need to perform additional tests on a line once it has matched a given search expression. Maybe we need to find a line that contains "good," "bad," and "ugly" in any order on the same line. Finding that kind of target string would be tough to do based on only the ideas that we have discussed so far in this chapter. One way to find such a combination of content would be with three `grep` commands in series as shown below.

```
grep "good" | grep "bad" | grep "ugly"
```

Such a one-liner would pass only those lines that contained all three words. If we wanted to do the same sort of thing with `awk` or `sed`, we could line up those commands in series the same as we did with `grep`:

```
awk '/good/' | awk '/bad/' | awk '/ugly/' # or
sed '/good/!d' | sed '/bad/!d' | sed '/ugly/!d'
```

That approach, although it works, seems rather blunt to me. What if we used the abilities in `awk` to perform all the necessary checks with one command?

```
awk '/good/ {if (($0~/bad/)&&($0~/ugly/)){print $0}}'
```

Or we could do all the checking with one explicit `if` statement as in

```
awk '{if (($0~/good/)&&($0~/bad/)&&($0~/ugly/)){print $0}}'
```

The above one-liner might not look like much, but consider this for a moment: the "OR" approach allows us to match any one of a number of conditions, and the "AND" approach allows us to match many combinations of conditions.

Handy Regular Expressions

Below are some expressions for matching things you might encounter in your scripting.

social security number: "###-##-####"

```
awk '/[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]/'
egrep '[0-9]{3}-[0-9]{2}-[0-9]{4}'
grep -E '[0-9]{3}-[0-9]{2}-[0-9]{4}'
grep '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]'
sed '/[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]/!d'
```

7-digit phone number: "###-####"

```
awk '/[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/'
egrep '[0-9]{3}-[0-9]{4}'
grep -E '[0-9]{3}-[0-9]{4}'
grep '[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]'
sed '/[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/!d'
```

10-digit phone number: "###-###-####"

```
awk '/[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/'
egrep '[0-9]{3}-[0-9]{3}-[0-9]{4}'
grep -E '[0-9]{3}-[0-9]{3}-[0-9]{4}'
grep '[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]'
sed '/[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/!d'
```

10-digit phone number with parentheses: "(###)###-####"

```
awk '/\([0-9][0-9][0-9]\)[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/'
egrep '\([0-9]{3}\)-[0-9]{3}-[0-9]{4}'
grep -E '\([0-9]{3}\)-[0-9]{3}-[0-9]{4}'
grep '\([0-9][0-9][0-9])[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]'
sed '/([0-9][0-9][0-9])[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/!d'
```

positive or negative integer number: "#####" (variable number of digits)

```
awk '/^[+-]?[0-9]+$/'
grep '/^[+-]?\?[0-9]\+$'
sed '/^[+-]?\?[0-9]\+$!/d'
```

positive or negative hexadecimal number: "#####" (variable number of digits)

```
awk '/^[+-]?[0-9A-Fa-f]+$/'
grep '/^[+-]?\?[0-9A-Fa-f]\+$'
sed '/^[+-]?\?[0-9A-Fa-f]\+$!/d'
```

decimal number: "###.###" (variable numbers of digits before and after decimal)

```
awk '/^[+-]?[0-9]+\.[0-9]+$/'
grep '/^[+-]?\?[0-9]\+\.[0-9]\+$'
sed '/^[+-]?\?[0-9]\+\.[0-9]\+$!/d'
```

To allow the absence of a digit after a decimal point as in "50." for the above example, use the expressions below

```
awk '/^[+-]?[0-9]+\.[0-9]*$/'
grep '/^[+-]?\?[0-9]\+\.[0-9]*$'
sed '/^[+-]?\?[0-9]\+\.[0-9]*$!/d'
```

To also allow the absence of a digit before a decimal point as in ".50" for the above example, use the expressions below

```
awk '/^[+-]?([0-9]+\.[0-9]*|\.[0-9]+)$/'
```

```
awk '/^[+-]?[0-9]+\.[0-9]*$/'
```

```
grep '/^[+-]?\?[0-9]\+\.[0-9]*$'
```

```
sed '/^[+-]?\?[0-9]\+\.[0-9]*$!/d'
```

integer or decimal number including those with leading or trailing decimals:

```
awk '/^[+-]?([0-9]+[.]?[0-9]*|[-+]?[.][0-9]+)$/'
```

```
grep -e '/^[+-]?\?[0-9]\+[.]\?[0-9]*$' -e '/[-+]\?[.][0-9]\+$'
```

```
grep '/^[+-]?\?([0-9]\+[.]\?[0-9]*|[-+]\?[.][0-9]\+$)'
```

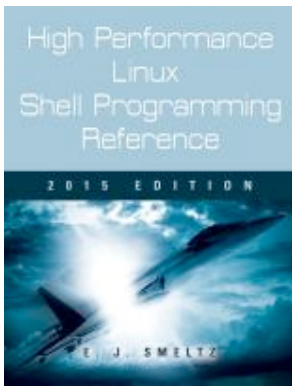
```
sed '/^[+-]?\?([0-9]\+[.]\?[0-9]*|[-+]\?[.][0-9]\+$)!/d'
```

exponential number:

```
awk '/^[+-]?([0-9]+[.]?[0-9]+)([ ]?[eE][+-]?[0-9]+)$/'  
grep '^[-+]\?\[0-9]\+\[.]\?[0-9]\+\)\([ ]?[eE][+-]\?[0-9]\+\)\$'  
sed '/^[+-]\?\[0-9]\+\[.]\?[0-9]\+\)\([ ]?[eE][+-]\?[0-9]\+\)\$!/d'
```

number of any format:

```
awk '/^[+-]?([0-9]+[.]?[0-9]*)([ ]?[eE][+-]?[0-9]+)?$/'  
grep '^[-+]\?\[0-9]\+\[.]\?[0-9]*\)\([ ]?[eE][+-]\?[0-9]\+\)\)?$'  
sed '/^[+-]\?\[0-9]\+\[.]\?[0-9]*\)\([ ]?[eE][+-]\?[0-9]\+\)\)?$!/d'
```



Extensive, example-based Linux shell programming reference includes an English-to-shell dictionary, a tutorial and handbook, and many tables of information useful to programmers. Besides listing more than 2000 shell one-liners, it explains the principles and techniques of how to increase performance (execution speed, reliability, and efficiency), which apply to many other programming languages beyond shell.

High Performance Linux Shell Programming Reference

2015 Edition

Order the complete book from

Booklocker.com

<http://www.booklocker.com/p/books/7831.html?s=pdf>

or from your favorite neighborhood
or online bookstore.