ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The AWK Programming Language

Second Edition

Alfred V. Aho
Brian W. Kernighan
Peter J. Weinberger

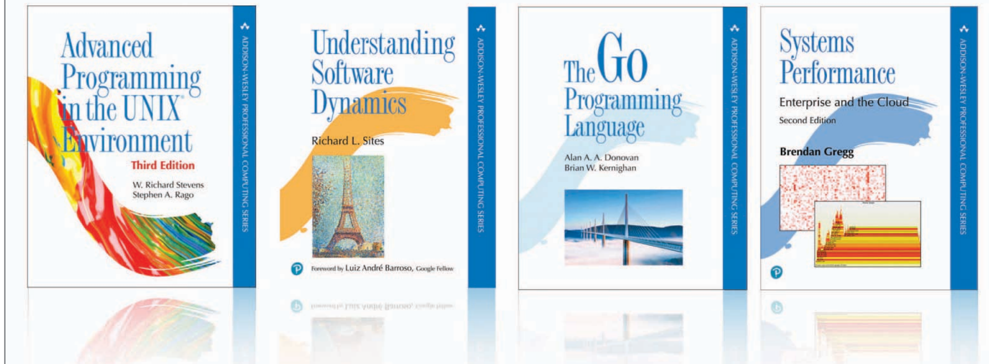
FREE SAMPLE CHAPTER

The AWK Programming Language

Second Edition

The Pearson Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor



Visit **informit.com/series/professionalcomputing**
for a complete list of available publications.

The **Pearson Addison-Wesley Professional Computing Series** was created in 1990 to provide serious programmers and networking professionals with well-written and practical reference books. Pearson Addison-Wesley is renowned for publishing accurate and authoritative books on current and cutting-edge technology, and the titles in this series will help you understand the state of the art in programming languages, operating systems, and networks.

Make sure to connect with us!
informit.com/connect

Pearson Addison-Wesley

informIT.com
the trusted technology learning source

The AWK Programming Language

Second Edition

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

Addison-Wesley

Hoboken, New Jersey

Cover image: "Great Auk" by John James Audubon from *The Birds of America, Vols. I-IV,* 1827–1838, Archives & Special Collections, University of Pittsburgh Library System

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2023941419

Copyright © 1988, 2024 Bell Telephone Laboratories, Incorporated.
UNIX is a registered trademark of The Open Group.

This book was formatted by the authors in Times Roman, Courier and Helvetica, using Groff, Ghostscript and other open source Unix tools. See https://www.awk.dev.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-826972-2
ISBN-10: 0-13-826972-6

$PrintCode

To the millions of Awk users

This page intentionally left blank

Contents

Preface ix

1. An Awk Tutorial 1
1.1 Getting Started 1
1.2 Simple Output 4
1.3 Formatted Output 7
1.4 Selection 8
1.5 Computing with Awk 10
1.6 Control-Flow Statements 13
1.7 Arrays 16
1.8 Useful One-liners 17
1.9 What Next? 19

2. Awk in Action 21
2.1 Personal Computation 21
2.2 Selection 23
2.3 Transformation 25
2.4 Summarization 27
2.5 Personal Databases 28
2.6 A Personal Library 31
2.7 Summary 34

3. Exploratory Data Analysis 35
3.1 The Sinking of the Titanic 36
3.2 Beer Ratings 41
3.3 Grouping Data 43
3.4 Unicode Data 45
3.5 Basic Graphs and Charts 47
3.6 Summary 49

4. Data Processing 51
4.1 Data Transformation and Reduction 51
4.2 Data Validation 57
4.3 Bundle and Unbundle 59
4.4 Multiline Records 60
4.5 Summary 66

5. Reports and Databases 67
5.1 Generating Reports 67
5.2 Packaged Queries and Reports 73
5.3 A Relational Database System 75
5.4 Summary 83

6. Processing Words 85
6.1 Random Text Generation 85
6.2 Interactive Text-Manipulation 90
6.3 Text Processing 92
6.4 Making an Index 99
6.5 Summary 105

7. Little Languages 107
7.1 An Assembler and Interpreter 108
7.2 A Language for Drawing Graphs 111
7.3 A Sort Generator 113
7.4 A Reverse-Polish Calculator 115
7.5 A Different Approach 117
7.6 A Recursive-Descent Parser for Arithmetic Expressions 119
7.7 A Recursive-Descent Parser for a Subset of Awk 122
7.8 Summary 126

8. Experiments with Algorithms 129
8.1 Sorting 129
8.2 Profiling 142
8.3 Topological Sorting 144
8.4 Make: A File Updating Program 148
8.5 Summary 153

9. Epilogue 155
9.1 Awk as a Language 155
9.2 Performance 157
9.3 Conclusion 160

Appendix A: Awk Reference Manual 163
A.1 Patterns 165
A.2 Actions 176
A.3 User-Defined Functions 196
A.4 Output 197
A.5 Input 202
A.6 Interaction with Other Programs 207
A.7 Summary 208

Index 209

Preface

Awk was created in 1977 as a simple programming language for writing short programs that manipulate text and numbers with equal ease. It was meant as a *scripting language* to complement and work well with Unix tools, following the Unix philosophy of having each program do one thing well and be composable with other programs.

The computing world today is enormously different from what it was in 1977. Computers are thousands of times faster and have a million times as much memory. Software is different too, with a rich variety of programming languages and computing environments. The Internet has given us more data to process, and it comes from all over the world. We're no longer limited to the 26 letters of English either; thanks to Unicode, computers process the languages of the world in their native character sets.

Even though Awk is nearly 50 years old, and in spite of the great changes in computing, it's still widely used, a core Unix tool that's available on any Unix, Linux, or macOS system, and usually on Windows as well. There's nothing to download, no libraries or packages to import — just use it. It's an easy language to learn and you can do a lot after only a few minutes of study.

Scripting languages were rather new in 1977, and Awk was the first such language to be widely used. Other scripting languages complement or sometimes replace Awk. Perl, which dates from 1987, was an explicit reaction to some of the limitations of Awk at the time. Python, four years younger than Perl, is by far the most widely used scripting language today, and for most users would be the natural next step for larger programs, especially to take advantage of the huge number of libraries in the Python ecosystem. On the web, and also for some standalone uses, JavaScript is the scripting language of choice. Other more niche languages are still highly useful, and "the shell" itself has become a variety of different shells with significantly enriched programming capabilities.

Programmers and other computer users spend a lot of time doing simple, mechanical data manipulation — changing the format of data, checking its validity, finding items that have some property, adding up numbers, printing summaries, and the like. All of these jobs ought to be mechanized, but it's a real nuisance to have to write a special-purpose program in a language like C or Python each time such a task comes up.

Awk is a programming language that makes it possible to handle simple computations with short programs, often only one or two lines long. An Awk program is a sequence of patterns and actions that specify what to look for in the input data and what to do when it's found. Awk searches a set of files that contain text (but not non-text formats like Word documents, spreadsheets, PDFs and so on) for lines that match any of the patterns; when a matching line is found, the corresponding action is performed. A pattern can select lines by combinations of regular expressions and comparison operations on strings, numbers, fields, variables, and array elements. Actions may perform arbitrary processing on selected lines; the action language looks like C but there are no declarations, and strings and numbers are built-in data types.

Awk scans text input files and splits each input line into fields automatically. Because so many things are automatic — input, field splitting, storage management, initialization — Awk programs are usually much shorter than they would be in a more conventional language. Thus one common use of Awk is for the kind of data manipulation suggested above. Programs, a line or two long, are composed at the keyboard, run once, then discarded. In effect, Awk is a general-purpose programmable tool that can replace a host of specialized tools or programs.

The same brevity of expression and convenience of operations make Awk valuable for prototyping larger programs. Start with a few lines, then refine the program until it does the desired job, experimenting with designs by trying alternatives quickly. Since programs are short, it's easy to get started, and easy to start over when experience suggests a different direction. And if necessary, it's straightforward to translate an Awk program into another language once the design is right.

Organization of the Book

The goal of this book is to teach you what Awk is and how to use it effectively. Chapter 1 is a tutorial on how to get started; after reading even a few pages, you will have enough information to begin writing useful programs. The examples in this chapter are short and simple, typical of the interactive use of Awk.

The rest of the book contains a variety of examples, chosen to show the breadth of applicability of Awk and how to make good use of its facilities. Some of the programs are ones we use personally; others illustrate ideas but are not intended for production use; a few are included just because they are fun.

Chapter 2 shows Awk in action, with a number of small programs that are derived from the way that we use Awk for our own personal programming. The examples are probably too idiosyncratic to be directly useful, but they illustrate techniques and suggest potential applications.

Chapter 3 shows how Awk can be used for exploratory data analysis: examining a dataset to figure out its properties, identify potential (and real) errors, and generally get a grip on what it contains before expending further effort with other tools.

The emphasis in Chapter 4 is on retrieval, validation, transformation, and summarization of data — the tasks that Awk was originally designed for. There is also a discussion of how to handle data like address lists that naturally comes in multiline chunks.

Awk is a good language for managing small, personal databases. Chapter 5 discusses the generation of reports from databases, and builds a simple relational database system and query language for data stored in multiple files.

Chapter 6 describes programs for generating text, and some that help with document preparation. One of the examples is an indexing program based on the one we used for this book.

Chapter 7 is about "little languages," that is, specialized languages that focus on a narrow domain. Awk is convenient for writing small language processors because its basic operations support many of the lexical and symbol table tasks encountered in translation. The chapter includes an assembler, a graphics language, and several calculators.

Awk is a good language for expressing certain kinds of algorithms. Because there are no declarations and because storage management is easy, an Awk program has many of the advantages of pseudo-code but Awk programs can be run, which is not true of pseudo-code. Chapter 8 discusses experiments with algorithms, including testing and performance evaluation. It shows several sorting algorithms, and culminates in a version of the Unix `make` program.

Chapter 9 explains some of the historical reasons why Awk is as it is, and contains some performance measurements, including comparisons with other languages. The chapter also offers suggestions on what to do when Awk is too slow or too confining.

Appendix A, the reference manual, covers the Awk language in a systematic order. Although there are plenty of examples in the appendix, like most manuals it's long and a bit dry, so you will probably want to skim it on a first reading.

You should begin by reading Chapter 1 and trying some small examples of your own. Then read as far into each chapter as your interest takes you. The chapters are nearly independent of each other, so the order doesn't matter much. Take a quick look at the reference manual to get an overview, concentrating on the summaries and tables, but don't get bogged down in the details.

The Examples

There are several themes in the examples. The primary one, of course, is to show how to use Awk well. We have tried to include a wide variety of useful constructions, and we have stressed particular aspects like associative arrays and regular expressions that typify Awk programming.

A second theme is to show Awk's versatility. Awk programs have been used from databases to circuit design, from numerical analysis to graphics, from compilers to system administration, from a first language for non-programmers to the implementation language for software engineering courses. We hope that the diversity of applications illustrated in the book will suggest new possibilities to you as well.

A third theme is to show how common computing operations are done. The book contains a relational database system, an assembler and interpreter for a toy computer, a graph-drawing language, a recursive-descent parser for an Awk subset, a file-update program based on `make`, and many other examples. In each case, a short Awk program conveys the essence of how something works in a form that you can understand and play with.

We have also tried to illustrate a spectrum of ways to attack programming problems. Rapid prototyping is one approach that Awk supports well. A less obvious strategy is divide and conquer: breaking a big job into small components, each concentrating on one aspect of the problem. Another is writing programs that create other programs. Little languages define a good user interface and may suggest a sound implementation. Although these ideas are presented here in the context of Awk, they are much more generally applicable, and ought to be

part of every programmer's repertoire.

The examples have all been tested directly from the text, which is in machine-readable form. We have tried to make the programs error-free, but they do not defend against all possible invalid inputs, so we can concentrate on conveying the essential ideas.

Evolution of Awk

Awk was originally an experiment in generalizing the Unix tools `grep` and `sed` to deal with numbers as well as text. It was based on our interests in regular expressions and programmable editors. As an aside, the language is officially AWK (all caps) after the authors' initials, but that seems visually intrusive, so we've used Awk throughout for the name of the language, and `awk` for the name of the program. (Naming a language after its creators shows a certain paucity of imagination. In our defense, we didn't have a better idea, and by coincidence, at some point in the process we were in three adjacent offices in the order Aho, Weinberger, and Kernighan.)

Although Awk was meant for writing short programs, its combination of facilities soon attracted users who wrote significantly larger programs. These larger programs needed features that had not been part of the original implementation, so Awk was enhanced in a new version made available in 1985.

Since then, several independent implementations of Awk have been created, including Gawk (maintained and extended by Arnold Robbins), Mawk (by Michael Brennan), Busybox Awk (by Dmitry Zakharov), and a Go version (by Ben Hoyt). These differ in minor ways from the original and from each other but the core of the language is the same in all. There are also other books about Awk, notably *Effective Awk Programming*, by Arnold Robbins, which includes material on Gawk. The Gawk manual itself is online, and covers that version very carefully.

The POSIX standard for Awk is meant to define the language completely and precisely. It is not particularly up to date, however, and different implementations do not follow it exactly.

Awk is available as a standard installed program on Unix, Linux, and macOS, and can be used on Windows through WSL, the Windows Subsystem for Linux, or a package like Cygwin. You can also download it in binary or source form from a variety of web sites. The source code for the authors' version is at `https://github.com/onetrueawk/awk`. The web site `https://www.awk.dev` is devoted to Awk; it contains code for all the examples from the book, answers to selected exercises, further information, updates, and (inevitably) errata.

For the most part, Awk has not changed greatly over the years. Perhaps the most significant new feature is better support for Unicode: newer versions of Awk can now handle data encoded in UTF-8, the standard Unicode encoding of characters taken from any language. There is also support for input encoded as comma-separated values, like those produced by Excel and other programs. The command

```
$ awk --version
```

will tell you which version you are running. Regrettably, the default versions in common use are sometimes elderly, so if you want the latest and greatest, you may have to download and install your own.

Since Awk was developed under Unix, some of its features reflect capabilities found in Unix and Linux systems, including macOS; these features are used in some of our examples.

Furthermore, we assume the existence of standard Unix utilities, particularly `sort`, for which exact equivalents may not exist elsewhere. Aside from these limitations, however, Awk should be useful in any environment.

Awk is certainly not perfect; it has its full share of irregularities, omissions, and just plain bad ideas. But it's also a rich and versatile language, useful in a remarkable number of cases, and it's easy to learn. We hope you'll find it as valuable as we do.

Acknowledgments

We are grateful to friends and colleagues for valuable advice. In particular, Arnold Robbins has helped with the implementation of Awk for many years. For this edition of the book, he found errors, pointed out inadequate explanations and poor style in Awk code, and offered perceptive comments on nearly every page of several drafts of the manuscript. Similarly, Jon Bentley read multiple drafts and suggested many improvements, as he did with the first edition. Several major examples are based on Jon's original inspiration and his working code. We deeply appreciate their efforts.

Ben Hoyt provided insightful comments on the manuscript based on his experience implementing a version of Awk in Go. Nelson Beebe read the manuscript with his usual exceptional thoroughness and focus on portability issues. We also received valuable suggestions from Dick Sites and Ozan Yigit. Our editor, Greg Doench, was a great help in every aspect of shepherding the book through Addison-Wesley. We also thank Julie Nahil for her assistance with production.

Acknowledgments for the First Edition

We are deeply indebted to friends who made comments and suggestions on drafts of this book. We are particularly grateful to Jon Bentley, whose enthusiasm has been an inspiration for years. Jon contributed many ideas and programs derived from his experience using and teaching Awk; he also read several drafts with great care. Doug McIlroy also deserves special recognition; his peerless talent as a reader greatly improved the structure and content of the whole book. Others who made helpful comments on the manuscript include Susan Aho, Jaap Akkerhuis, Lorinda Cherry, Chris Fraser, Eric Grosse, Riccardo Gusella, Bob Herbst, Mark Kernighan, John Linderman, Bob Martin, Howard Moscovitz, Gerard Schmitt, Don Swartwout, Howard Trickey, Peter van Eijk, Chris Van Wyk, and Mihalis Yannakakis. We thank them all.

Alfred V. Aho
Brian W. Kernighan
Peter J. Weinberger

This page intentionally left blank

3

Exploratory Data Analysis

The previous chapter described a number of small scripts for personal use, often idiosyncratic or specialized. In this chapter, we're going to do something that is also typical of how Awk is used in real life: we'll use it along with other tools to informally explore some real data, with the goal of seeing what it looks like. This is called *exploratory data analysis* or *EDA*, a term first used by the pioneering statistician John Tukey.

Tukey invented a number of basic data visualization techniques like boxplots, inspired the statistical programming language S that led to the widely-used R language, co-invented the Fast Fourier Transform, and coined the words "bit" and "software." The authors knew John Tukey as a friend and colleague at Bell Labs in the 1970s and 1980s, where among a large number of very smart and creative people, he stood out as someone special.

The essence of exploratory data analysis is to play with the data before making hypotheses or drawing conclusions. As Tukey himself said,

> "Finding the question is often more important than finding the answer. Exploratory data analysis is an attitude, a flexibility, and a reliance on display, NOT a bundle of techniques."

In many cases, that involves counting things, computing simple statistics, arranging data in different ways, looking for patterns, commonalities, outliers and oddities, and drawing basic graphs and other visual displays. The emphasis is on small, quick experiments that might give some insight, rather than polish or refinement; those come later when we have a better sense of what the data might be telling us.

For EDA, we typically use standard Unix tools like the shell, `wc`, `diff`, `sort`, `uniq`, `grep`, and of course regular expressions. These combine well with Awk, and often with other languages like Python.

We will also encounter a variety of file formats, including comma- or tab-separated values (CSV and TSV), JSON, HTML, and XML. Some of these, like CSV and TSV, are easily processed in Awk, while others are sometimes better handled with other tools.

3.1 The Sinking of the Titanic

Our first dataset is based on the sinking of the Titanic on April 15, 1912. This example was chosen, not entirely by coincidence, by one of the authors, who was at the time on a trans-Atlantic boat trip, passing not far from the site where the Titanic sank.

Summary Data: titanic.tsv

The file `titanic.tsv`, adapted from Wikipedia, contains summary data about the Titanic's passengers and crew. As is common with datasets in CSV and TSV format, the first line is a header that identifies the data in the lines that follow. Columns are separated by tabs.

```
Type    Class   Total   Lived   Died
Male    First   175     57      118
Male    Second  168     14      154
Male    Third   462     75      387
Male    Crew    885     192     693
Female  First   144     140     4
Female  Second  93      80      13
Female  Third   165     76      89
Female  Crew    23      20      3
Child   First   6       5       1
Child   Second  24      24      0
Child   Third   79      27      52
```

Many (perhaps all) datasets contain errors. As a quick check here, each line should have five fields, and the total in the third field should equal field four (lived) plus field five (died). This program prints any line where those conditions do not hold:

```
NF != 5 || $3 != $4 + $5
```

If the data is in the right format and the numbers are correct, this should produce a single line of output, the header:

```
Type    Class   Total   Lived   Died
```

Once we've done this minimal check, we can look at other things. For example, how many people are there in each category?

The categories that we want to count are not identified by numbers, but by words like `Male` and `Crew`. Fortunately, the subscripts or indices of Awk arrays can be arbitrary strings of characters, so `gender["Male"]` and `class["Crew"]` are valid expressions.

Arrays that allow arbitrary strings as subscripts are called *associative arrays*; other languages provide the same facility with names like dictionary, map or hashmap. Associative arrays are remarkably convenient and flexible, and we will use them extensively.

```
NR > 1 { gender[$1] += $3; class[$2] += $3 }

END {
  for (i in gender) print i, gender[i]
  print ""
  for (i in class) print i, class[i]
}
```

gives

```
Male 1690
Child 109
Female 425

Crew 908
First 325
Third 706
Second 285
```

Awk has a special form of the `for` statement for iterating over the indices of an associative array:

`for (`*i* `in` *array*`) {` *statements* `}`

sets the variable *i* in turn to each index of the array, and the *statements* are executed with that value of *i*. The elements of the array are visited in an unspecified order; you can't count on any particular order.

What about survival rates? How did social class, gender and age affect the chance of survival among passengers? With this summary data we can do some simple experiments, for example, computing the survival rate for each category.

```
NR > 1 { printf("%6s  %6s  %6.1f%%\n", $1, $2, 100 * $4/$3) }
```

We can sort the output of this test by piping it through the Unix command `sort -k3 -nr` (sort by third field in reverse numeric order) to produce

```
 Child  Second   100.0%
Female   First    97.2%
Female    Crew    87.0%
Female  Second    86.0%
 Child   First    83.3%
Female   Third    46.1%
 Child   Third    34.2%
  Male   First    32.6%
  Male    Crew    21.7%
  Male   Third    16.2%
  Male  Second     8.3%
```

Evidently women and children did survive better on average.

Note that these examples treat the header line of the dataset as a special case. If you're doing a lot of experiments, it may be easier to remove the header from the data file than to ignore it explicitly in every program.

Passenger Data: passengers.csv

The file `passengers.csv` is a larger file that contains detailed information about passengers, though it does not contain anything about crew members. The original file is a merger of a widely used machine-learning dataset with another list from Wikipedia. It has 11 columns including home town, lifeboat assignment, and ticket price:

```
"row.names","pclass","survived","name","age","embarked",
   "home.dest","room","ticket","boat","sex"
...
"11","1st",0,"Astor, Colonel John Jacob",47,"Cherbourg",
   "New York, NY","","17754 L224 10s 6d","(124)","male"
...
```

How big is the file? We can use the Unix `wc` command to count lines, words and characters:

```
$ wc passengers.csv
    1314    6794  112466 passengers.csv
```

or a two-line Awk program like the one we saw in Chapter 1:

```
     { nc += length($0) + 1; nw += NF }
END { print NR, nw, nc, FILENAME }
```

Except for spacing, they produce the same results when the input is a single file.

The file format of `passengers.csv` is comma-separated values. Although CSV is not rigorously defined, one common definition says that any field that contains a comma or a double quote (`"`) must be surrounded by double quotes. Any field may be surrounded by quotes, whether it contains commas and quotes or not. An empty field is just `""`, and a quote within a field is represented by a doubled quote, as in `""","""`, which represents `","`. Input fields in CSV files may contain newline characters. For more details, see Section A.5.2.

This is more or less the format used by Microsoft Excel and other spreadsheet programs like Apple Numbers and Google Sheets. It is also the default input format for data frames in Python's Pandas library and in R.

In versions of Awk since 2023, the command-line argument `--csv` causes input lines to be split into fields according to this rule. Setting the field separator to a comma explicitly with `FS=,` does not treat comma field separators specially, so this is useful only for the simplest form of CSV: no quotes. With older versions of Awk it may be easiest to convert the data to a different form using some other system, like an Excel spreadsheet or a Python CSV module.

Another useful alternative format is tab-separated values or TSV. The idea is the same, but simpler: fields are separated by single tabs, and there is no quoting mechanism so fields may not contain embedded tabs or newlines. This format is easily handled by Awk, by setting the field separator to a tab with `FS="\t"` or equivalently with the command-line argument `-F"\t"`.

As an aside, it's wise to verify whether a file is in the proper format before relying on its contents. For example, to check whether all records have the same number of fields, you could use

```
awk '{print NF}' file | sort | uniq -c | sort -nr
```

The first `sort` command brings all instances of a particular value together; then the command `uniq -c` replaces each sequence of identical values by a single line with a count and the value; and finally `sort -nr` sorts the result numerically in reverse order, so the largest values come first.

For `passengers.csv`, using the `--csv` option to process CSV input properly, this produces

```
1314 11
```

Every record has the same number of fields, which is necessary for valid data in this dataset, though not sufficient. If some lines have different numbers of fields, now use Awk to find them, for example with `NF != 11` in this case.

With a version of Awk that does not handle CSV, the output using `-F,` will be different:

```
 624 12
 517 13
 155 14
  15 15
   3 11
```

This shows that almost all fields contain embedded commas.

By the way, generating CSV is straightforward. Here's a function `to_csv` that converts a string to a properly quoted string by doubling each quote and surrounding the result with quotes. It's an example of a function that could go into a personal library.

```
# to_csv - convert s to proper "..."

function to_csv(s) {
  gsub(/"/, "\"\"", s)
  return "\"" s "\""
}
```

(Note how quotes are quoted with backslashes.)

We can use this function within a loop to insert commas between elements of an array to create a properly formatted CSV record for an associative array, or for an indexed array like the fields of a line, as illustrated in the functions `rec_to_csv` and `arr_to_csv`:

```
# rec_to_csv - convert a record to csv

function rec_to_csv(   s, i) {
  for (i = 1; i < NF; i++)
    s = s to_csv($i) ","
  s = s to_csv($NF)
  return s
}

# arr_to_csv - convert an indexed array to csv

function arr_to_csv(arr,   s, i, n) {
  n = length(arr)
  for (i = 1; i <= n; i++)
    s = s to_csv(arr[i]) ","
  return substr(s, 1, length(s)-1) # remove trailing comma
}
```

The following program selects the five attributes class, survival, name, age, and gender, from the original file, and converts the output to tab-separated values.

```
NR > 1 { OFS="\t"; print $2, $3, $4, $5, $11 }
```

It produces output like this:

```
1st 0   Allison, Miss Helen Loraine  2  female
1st 0   Allison, Mr Hudson Joshua Creighton 30  male
1st 0   Allison, Mrs Hudson J.C. (Bessie Waldo Daniels) 25  female
1st 1   Allison, Master Hudson Trevor    0.9167 male
```

Most ages are integers, but a handful are fractions, like the last line above. Helen Allison was two years old; Master Hudson Allison appears to have been 11 months old, and was the only survivor in his family. (From other sources, we know that the Allison's chauffeur, George Swane, age 18, also died, but the family's maid and cook both survived.)

How many infants were there? Running the command

```
$4 < 1
```

with tab as the field separator produces eight lines:

```
1st 1   Allison, Master Hudson Trevor    0.9167 male
2nd 1   Caldwell, Master Alden Gates     0.8333 male
2nd 1   Richards, Master George Sidney   0.8333 male
3rd 1   Aks, Master Philip   0.8333 male
3rd 0   Danbom, Master Gilbert Sigvard Emanuel   0.3333 male
3rd 1   Dean, Miss Elizabeth Gladys (Millvena)   0.1667 female
3rd 0   Peacock, Master Alfred Edward   0.5833  male
3rd 0   Thomas, Master Assad Alexander  0.4167  male
```

Exercise 3-1. Modify the word count program to produce a separate count for each of its input files, as the Unix `wc` command does. □

Some Further Checking

Another set of questions to explore is how well the two data sources agree. They both come from Wikipedia, but it is not always a perfectly accurate source. Suppose we check something absolutely basic, like how many passengers there were in the `passengers` file:

```
$ awk 'END {print NR}' passengers.csv
1314
```

This count includes one header line, so there were 1313 passengers. On the other hand, this program adds up the counts for non-crew members from the third field of the summary file:

```
$ awk '!/Crew/ { s += $3 }; END { print s }' titanic.tsv
1316
```

That's a discrepancy of three people, so something is wrong.

As another example, how many children were there?

```
awk --csv '$5 <= 12' passengers.csv
```

produces 100 lines, which doesn't match the 109 children in `titanic.tsv`. Perhaps children are those 13 or younger? That gives 105. Younger than 14? That's 112. We can guess what age is being used by counting passengers who are called "Master":

```
awk --csv '/Master/ {print $5}' passengers.csv | sort -n
```

The largest age in this population is 13, so that's perhaps the best guess, though not definitive.

In both of these cases, numbers that ought to be the same are in fact different, which suggests that the data is still flaky. When exploring data, you should always be prepared for

errors and inconsistencies in form and content. A big part of the job is to be sure that you have identified and dealt with potential problems before starting to draw conclusions.

In this section, we've tried to show how simple computations can help identify such problems. If you collect a set of tools for common operations, like isolating fields, grouping by category, printing the most common and least common entries, and so on, you'll be better able to perform such checks.

Exercise 3-2. Write some of these tools for yourself, according to your own needs and tastes. □

3.2 Beer Ratings

Our second dataset is a collection of nearly 1.6 million ratings of beer, originally from RateBeer.com, a site for beer enthusiasts. This dataset is so large that it's not feasible to study every line to be sure of its properties, so we have to rely on tools like Awk to explore and validate the data.

The data comes from Kaggle, a site for experimenting with machine-learning algorithms. You can find the original at `https://www.kaggle.com/datasets/rdoume/beerreviews`; we are grateful to RateBeer, Kaggle, and the creator of the dataset itself for providing such an interesting collection of data.

Let's start with some of the basic parameters: how big is the file and what does it look like? For a raw count, nothing beats the `wc` command:

```
$ time wc reviews.csv
 1586615 12171013 180174429 reviews.csv
real    0m0.629s
user    0m0.585s
sys     0m0.037s
```

Not surprisingly, `wc` is fast but as we've seen before, it's easy to write a `wc` equivalent in Awk:

```
$ time awk '{ nc += length($0) + 1; nw += NF }
END { print NR, nw, nc, FILENAME }' reviews.csv
1586615 12170527 179963813 reviews.csv
real    0m9.402s
user    0m9.159s
sys     0m0.125s
```

Awk is an order of magnitude slower for this specific test. Awk is fast enough for most purposes, but there are times when other programs are more appropriate. Somewhat surprisingly, Gawk is five times faster, taking only 1.9 seconds.

Something else is more surprising, however: `wc` and Awk differ in the number of words and characters they count. We'll dig into this later, but as a preview, `wc` is counting bytes (and thus implicitly assuming that the input is entirely ASCII), while Awk is counting Unicode UTF-8 characters. Here's an example rating where the two programs come up with legitimately different answers:

```
95,Löwenbräu AG,1257106630,4,4,3,atis,Munich Helles Lager,4,4,
      Löwenbräu Urtyp,5.4,33038
```

UTF-8 is a variable-length encoding: ASCII characters are a single byte, and other languages use two or three bytes per character. The characters with umlauts are two bytes long in UTF-8. There are also some records with Asian characters, which are three bytes long. In

such cases, wc will report more characters than Awk will.

The original data has 13 attributes but we will only use five of them here: brewery name, overall review, beer style, beer name, and alcohol content (percentage of alcohol by volume, or ABV). We created a new file with these attributes, and also converted the format from its original CSV to TSV by setting the output field separator OFS. This produces lines like this. (Long lines have been split into two, marked by a backslash at the end.)

```
Amstel Brouwerij B. V.  3.5 Light Lager Amstel Light    3.5
Bluegrass Brewing Co.   4   American Pale Ale (APA) American \
     Pale Ale   5.79
Hoppin' Frog Brewery    2.5 Winter Warmer   Frosted Frog \
     Christmas Ale  8.6
```

This shrinks the file from 180 megabytes to 113 megabytes, still large but more manageable.

We can see a wide range of ABV values in these sample lines, which suggests a question: What's the maximum value, the strongest beer that has been reviewed? This is easily answered with this program:

```
NR > 1 && $5 > maxabv { maxabv = $5; brewery = $1; name = $4 }
END { print maxabv, brewery, name }
```

which produces

```
57.7 Schorschbräu Schorschbräu Schorschbock 57%
```

This value is stunningly high, about 10 times the content of normal beer, so on the surface it looks like a data error. But a trip to the web confirms its legitimacy. That raises a follow-up question, whether this value is a real outlier, or merely the tip of a substantial alcoholic iceberg. If we look for brews of say 10 percent or more:

```
$5 >= 10 { print $1, $4, $5 }
```

we get over 195,000 reviews, which suggests that high-alcohol beer is popular, at least among people who contribute to RateBeer.

Of course that raises yet more questions, this time about low-alcohol beer. What about beer with less than say 0.5 percent, which is the legal definition of alcohol-free, at least in parts of the USA?

```
$5 <= 0.5 { print $1, $4, $5 }
```

This produces only 68,800 reviews, which suggests that low-alcohol beer is significantly less popular.

What ratings are associated with high and low alcohol?

```
$ awk -F'\t' '$5 >= 10 {rate += $2; nrate++}
    END {print rate/nrate, nrate}' rev.tsv
3.93702 194359

$ awk -F'\t' '$5 <= 0.5 {rate += $2; nrate++}
    END {print rate/nrate, nrate}' rev.tsv
3.61408 68808
```

```
$ awk -F'\t' '{rate += $2; nrate++}
    END {print rate/nrate, nrate}' rev.tsv
3.81558 1586615
```

This may or may not be statistically significant, but the average rating of high-alcohol beers is higher than the overall average rating, which in turn is higher than low-alcohol beers. (This is consistent with the personal preferences of at least one of the authors.)

But wait! Further checking reveals that there are 67,800 reviews that don't list an ABV at all; the field is empty! Let's re-run the low-alcohol computation with a proper test:

```
$ awk -F'\t' '$5 != "" && $5 <= 0.5 {rate += $2; nrate++}
    END {print rate/nrate, nrate}' rev.tsv
2.58895 1023
```

One doesn't have to be a beer aficionado to guess that beer without alcohol isn't going to be popular or highly rated.

The moral of these examples is that one has to look at all the data carefully. How many fields are empty or have an explicitly non-useful value like "N/A"? What is the range of values in a column? What are the distinct values? Answering such questions should be part of the initial exploration, and creating some simple scripts to automate the process can be a good investment.

3.3 Grouping Data

Let's take a look at the question of how many distinct values there are in a dataset. The sequence we showed above with `sort` and `uniq -c` is run so frequently that it probably ought to be a script, though at this point we've used it so many times that we can type it quickly and accurately. Here are some "distinct value" questions for the Titanic data, which we'll use because it's smaller.

How many male passengers and female passengers are there?

```
$ awk --csv '{g[$11]++}
     END {for (i in g) print i, g[i]}' passengers.csv
female 463
sex 1
male 850
```

That seems right — "sex" is the column header, and all the other values are either male or female, as expected. Very similar programs could check passenger classes, survival status, and age. For instance, checking ages reveals that no age is given for 258 of the 1313 passengers.

If we count the number of different ages with

```
$ awk --csv '{g[$5]++}
      END {for (i in g) print i, g[i]}' passengers.csv | sort -n
```

we see a sequence of lines like this:

```
...
 1 4
1 4
 2 6
2 7
 3 6
3 2
...
```

About half of the age fields contain a spurious space! That could easily throw off some future computation if it's not corrected.

More generally, sorting is a powerful technique for spotting anomalies in data, because it brings together pieces of text that share a common prefix but differ thereafter. We can see an example if we try to count honorifics, like *Mr* or *Colonel*. A quick list can be produced by printing the second word of the name field; this catches most of the obvious ones:

```
$ awk --csv '{split($4, name, " ")
   print name[2]}' passengers.csv | sort | uniq -c | sort -nr
 728 Mr
 229 Miss
 191 Mrs
  56 Master
  16 Ms
   7 Dr
   6 Rev
   ...
$
```

This produces a long tail of spurious non-honorifics, but also suggests places where the program could be improved; for example, removing punctuation would eliminate these differences:

```
   6 Rev
   1 Rev.
   1 Mlle.
   1 Mlle
```

This experiment also reveals one `Colonel` and one `Col`, presumably both referring to the same rank.

It's also interesting that *Ms* was in use more than 50 years before it became common in modern times, though we don't know what social status or condition it was meant to indicate.

In a similar vein, we can answer questions like how many breweries, beer styles, and reviewers are in the beer dataset:

```
{ brewery[$2]++; style[$8]++; reviewer[$7]++ }
END { print length(brewery), "breweries," length(style), "styles,"
            length(reviewer), "reviewers" }
```

produces

```
5744 breweries, 105 styles, 33389 reviewers
```

When applied to an array, the function `length` returns the number of elements.

Variations of this code can answer questions like how popular the various styles are:

```
{ style[$8]++ }
END { for (i in style) print style[i], i }
```

yields (when sorted and run through the head and tail program of Section 2.2)

```
117586 American IPA
85977 American Double / Imperial IPA
63469 American Pale Ale (APA)
54129 Russian Imperial Stout
50705 American Double / Imperial Stout
...
686 Gose
609 Faro
466 Roggenbier
297 Kvass
241 Happoshu
```

If you're going to do much of this kind of selecting fields and computing their statistics, it might be worth writing a handful of short scripts, rather like those we talked about in Chapter 2. One script could select a particular field, while a separate script could do the sorting and uniquing.

3.4 Unicode Data

As befits a drink that knows no national boundaries, the names of beers use many non-ASCII characters. The Awk program `charfreq` counts the number of times each distinct Unicode code point occurs in the input. (A code point is often a character, but some characters are made up of multiple code points.)

```
# charfreq - count frequency of characters in input

awk '
{ n = split($0, ch, "")
  for (i = 1; i <= n; i++)
    tab[ch[i]]++
}

END {
  for (i in tab)
    print i "\t" tab[i]
} ' $* | sort -k2 -nr
```

Splitting each line with an empty string as the field separator puts each character into a separate element of an array `ch`, and those characters are counted in `tab`; the accumulated counts are displayed at the end, sorted into decreasing frequency order.

This program is not very fast on this data, taking 250 seconds on a 2015 MacBook Air. Here's an alternate version that's more than twice as fast, just under 105 seconds:

```
# charfreq2 - alternate version of charfreq

awk '
{ n = length($0)
  for (i = 1; i <= n; i++)
    tab[substr($0, i, 1)]++
}

END {
  for (i in tab)
    print i "\t" tab[i]
} ' $* | sort -k2 -nr
```

Rather than using `split`, it extracts the characters one at a time with `substr`. The substring function `substr(`*s*`,`*m*`,`*n*`)` returns the substring of *s* of length *n* that begins at position *m* (starting at 1), or the empty string if the range implied by *m* and *n* is outside the string. If *n* is omitted, the substring extends to the end of *s*. Full details are in Section A.2.1 of the reference manual.

Gawk, the GNU version of Awk, is again much faster: 72 seconds for the first version and 42 seconds for the second.

What about another language? For comparison, we wrote a simple Python version of `charfreq`:

```
# charfreq - count frequency of characters in input

freq = {}
with open('../beer/reviews.csv', encoding='utf-8') as f:
  for ch in f.read():
    if ch == '\n':
      continue
    if ch in freq:
      freq[ch] += 1
    else:
      freq[ch] = 1
for ch in freq:
  print(ch, freq[ch])
```

The Python version takes 45 seconds, so it's about the same as Gawk, at the price of having to write explicit file-handling code. (The authors are not Pythonistas, so this program can surely be improved.)

There are 195 distinct characters in the file, excluding the newline at the end of each line. The most frequent character is a space, followed by printable characters:

```
        10586176
,       19094985
e       12308925
r       8311408
4       7269630
a       7014111
5       6993858
...
```

There are quite a few characters from European languages, like umlauts from German, and a modest number of Japanese and Chinese characters:

```
ア      1
ケ      1
サ      1
ル      1
山      1
葉      1
黑      229
```

The final character is 黑 (hēi, black), which appears in the name of a potent Imperial stout called simply "Black," with the Chinese character as its alternate name:

```
Mikkeller ApS,2,American Double / Imperial Stout,Black (黑),17.5
```

3.5 Basic Graphs and Charts

Visualization is an important component of exploratory data analysis, and fortunately there are really good plotting libraries that make graphs and charts remarkably easy. This is especially true of Python, with packages like Matplotlib and Seaborn, but Gnuplot, which is available on Unix and macOS, is also good for quick plotting. And of course Excel and other spreadsheet programs create good charts. We're not going to do much more here than to suggest minimal ways to plot data; after that, you should do your own experiments.

Is there a correlation between ABV and rating? Do reviewers prefer higher-alcohol beer? A scatter plot is one way to get a quick impression, but it's hard to plot 1.5 million points. Let's use Awk to grab a 0.1% sample (about 1,500 points), and plot that:

```
$ awk -F'\t' 'NR%1000 == 500 {print $2, $5}' rev.tsv >temp
$ gnuplot
plot 'temp'
$
```

This produces the graph in Figure 3-1. There appears to be at most a weak correlation between rating and ABV.

Tukey's boxplot visualization shows the median, quartiles, and other properties of a dataset. A boxplot is sometimes called a box and whiskers plot because the "whiskers" at each end of the box extend from the box typically by one and a half times the range between the lower and upper quartile. Points beyond the whiskers are outliers.

This short Python program generates a boxplot of beer ratings for the sample described above. The file `temp` contains the ratings and ABV, one pair per line, separated by a space, with no heading.

```
import matplotlib.pyplot as plt
import pandas as pd
df = pd.read_csv('temp', sep=' ', header=None)
plt.boxplot(df[0])
plt.show()
```

It produces the boxplot of Figure 3-2, which shows that the median rating is 4, and half the ratings are between the quartiles of 3.5 and 4.5. The whiskers extend to at most 1.5 times the inter-quartile range, and there are outliers at 1.5 and 1.0.

It's also possible to see how well any particular beer or brewery does, perhaps in comparison to mass-market American beers:

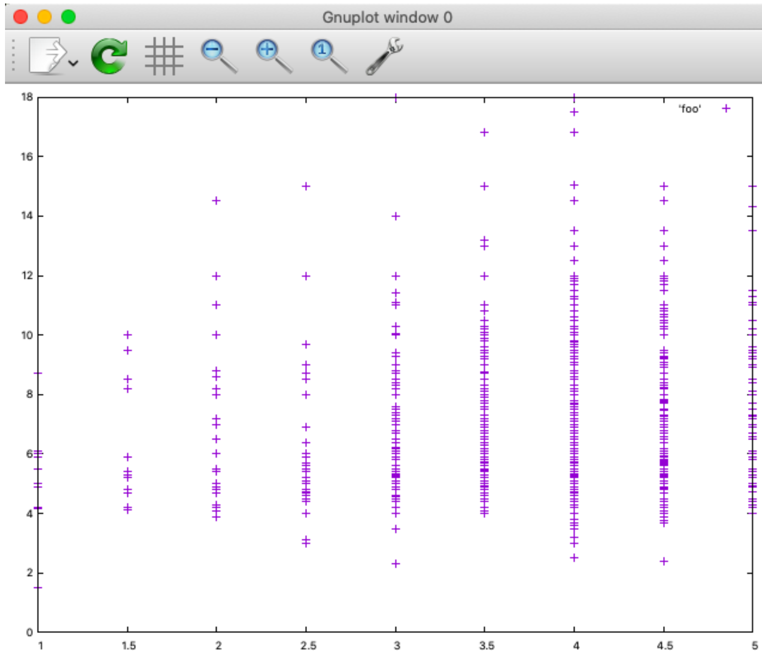


Figure 3-1: Beer rating as a function of ABV

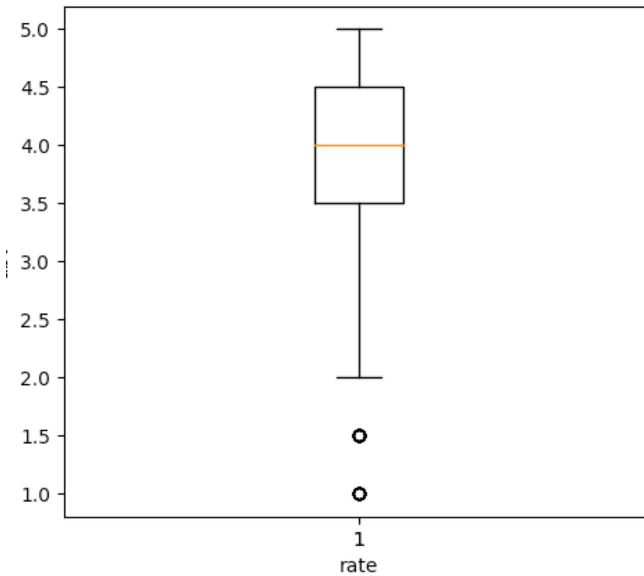


Figure 3-2: Boxplot of beer ratings sample.

```
$ awk -F'\t' '/Budweiser/ { s += $2; n++ }
      END {print s/n, n }' rev.tsv
3.15159 3958

$ awk -F'\t' '/Coors/ { s += $2; n++ }
      END {print s/n, n }' rev.tsv
3.1044 9291

$ awk -F'\t' '/Hill Farmstead/ { s += $2; n++ }
      END {print s/n, n }' rev.tsv
4.29486 1555
```

This suggests a significant ratings gap between mass-produced beers and small-scale craft brews.

3.6 Summary

The purpose of exploratory data analysis is to get a sense of what the data is, looking for both patterns and anomalies, before hypothesizing about results. As John Tukey said,

> The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.
>
> Be approximately right rather than exactly wrong.
>
> Far better an approximate answer to the right question, which is often vague, than the exact answer to the wrong question, which can always be made precise.

Awk is well worth learning as a core tool for exploratory data analysis, because you can use it for quick counting, summarization, and searching. It certainly won't handle everything, but in conjunction with other tools, especially spreadsheets and plotting libraries, it's excellent for getting a quick understanding of what a dataset contains.

A big part of this is to identify anomalies and weirdnesses. As a colleague at Bell Labs once told us long ago, "a third of all data is bad." Although he perhaps exaggerated for rhetorical effect, we have seen plenty of examples of datasets where a significant part really was flaky and untrustworthy. If you build a set of tools and techniques for looking at your data, you'll be better able to find the places where it needs to be cleaned up or at least treated cautiously.

This page intentionally left blank

Index

_ underscore 177
! NOT operator 9, 169, 181
!~ nonmatch operator 167, 170, 174, 181–182
"..." string constant 6, 167, 177, 194
comment 14, 164
#include processor 205, 207
$ regular expression 92, 172
$0 at end of input 12
$0 blank line 188
$0 record variable 5, 178
$0, side-effects on 179, 186
$*n* field 5, 178
% format conversion 184
% remainder operator 15, 181, 189
%% in printf 59
%= assignment operator 180
& in substitution 55, 185
&& AND operator 9, 134, 169, 181
'' quotes 2, 4, 207
() regular expression 172
* format conversion 99
* regular expression 173
*= assignment operator 180
+ regular expression 173
++ increment operator 11, 134, 181
+= assignment operator 180
- in character class 172
- standard input filename 91, 164, 204–205
-- decrement operator 11, 53, 86, 182
-- end of command-line options 206
-- option 164
--csv option 33, 38, 164, 166, 195, 203, 205
--version option xii, 164, 205
-= assignment operator 180
-F option 164, 203, 205
-f option 4, 32, 164, 205, 207
-f options, multiple 164
-v option 205
. regular expression 172
/= assignment operator 180
/dev/stderr 92, 123, 201
/dev/stdin 201
/dev/stdout 123, 201
/dev/tty file 201
= assignment operator 179
== comparison operator 9, 188
> comparison operator 8
> output redirection 199–200
>= comparison operator 8
>> output redirection 199–200
>>*file*, print 197
>*file*, print 69, 197
? regular expression 173
?: conditional expression 52, 180
[:alpha:] character class 94
[:punct:] character class 94
[^...] regular expression 172
\ backslash 31, 39, 171, 174, 183, 185
π, computation of 182
\033 escape character 173
\b backspace character 173
\n newline character 7, 59, 173
\t tab character 14, 166, 173
\u Unicode escape 173
\x hexadecimal escape 173
^ exponentiation operator 14, 181, 189
^ regular expression 92, 172
^= assignment operator 180
{...} braces 14, 142, 164, 190
{} regular expression 173
| input pipe 33, 205
| output redirection 201
| regular expression 169, 172
|*file*, print 197
|| OR operator 9, 169, 181
~ match operator 167, 170, 174, 181–182

action, default 5, 8, 163
actions, summary of 176
add checks and deposits 65
addcomma program 54
address list 60–61
address list, sorting 62
addup program 28
addup2 program 52
addup3 program 52
aggregation 93, 194, 201
Aho, A. V. 119, 155
Aho, S. xiii
Akkerhuis, J. xiii
algorithm, depth-first search 147, 151
  heapsort 137
  insertion sort 129
  linear 133, 157
  make update 151
  n log *n* 137, 140
  quadratic 133, 137, 157
  quicksort 135
  random permutation 87
  random selection 86
  topological sort 145
AND operator, && 9, 134, 169, 181
ARGC variable 23, 178, 206
arguments, command-line 206
arguments, function 197
ARGV variable 22–23, 91, 178, 206–208
ARGV, changing 91, 207–208
arith program 91
arithmetic expression grammar 120
arithmetic functions, table of 182
arithmetic operators 181, 186
arithmetic operators, table of 189
array parameter 197
array reference, cost of 158
array subscripts 193–195
array, associative 36, 193–194
arrays 16, 193
arrays, multidimensional 81, 89, 156, 195
asm program 110
assembler instructions, table of 108
assembly language 109
assignment expression 28, 102, 180
assignment operator, %= 180
  *= 180
  += 180
  -= 180

/= 180
= 179
^= 180
assignment operators 180
assignment, multiple 180
assignment, side-effects of 186
associative array 36, 193–194
associativity of operators 189
atan2 function 182
Avogadro's number 177
avoiding sort options 69, 94, 113
Awk command line 1, 3, 164, 205, 207
Awk grammar 122
Awk program, form of 2, 163
Awk program, running an 3
Awk programs, running time of 158–159
Awk versions xii, 38, 157, 164, 205
awk.dev xii
awk.parser program 124

back edge 147–148
backslash, \ 31, 39, 171, 174, 183, 185
backspace character, \b 173
bailing out 4
balanced delimiters 57
base and derived tables 79
batch sort test program 131
BeautifulSoup Python package 31
Beebe, N. xiii
BEGIN and END, multiple 143, 166
BEGIN pattern 10, 166, 206
Bentley, J. L. xiii, 99, 108, 113, 117, 153
binary tree 138
blank line separator 61
blank line, $0 188
blank line, printing a 10, 198
bluebird of happiness 173
bmi program 21
body mass index (BMI) 21
boundary condition testing 131
boxplot 35, 47
braces, {...} 14, 142, 164, 190
breadth-first order 138, 145
break statement 192
Brennan, M. xii
Budweiser 49
built-in variables, table of 178
bundle program 60
Busybox Awk xii

calc1 program 116
calc2 program 117
calc3 program 121
call by reference 197
call by value 197
capitals file 76
cat command 201, 207
cf program 22
changing ARGV 91, 207–208
character class, – in 172
complemented 172
named 94, 171
regular expression 172
[:alpha:] 94
[:punct:] 94
characters, table of escape 173
charfreq program 158
check function 132
check1 program 65
check2 program 65
check3 program 66
checkgen program 59
checking, cross-reference 56
checkpasswd program 58
checks and deposits, add 65
Cherry, L. L. xiii
chmod command 208
cliche program 87
close function 60, 202
coercion rules 186
coercion, number to string 130, 156, 167, 186
coercion, string to number 156, 167, 186
colcheck program 57
columns, summing 51
comma, line continuation after 164
comma-separated values xii, 38, 166, 203
command interpreter, shell 4, 207
command line, Awk 1, 3, 164, 205, 207
command, cat 201, 207
chmod 208
curl 31
date 33, 205
egrep 155, 159
gcc 150
grep xii, 155, 159, 202
join 77
ls 151
make 148
nm 56
pr 150
ptx 98
sed xii, 155, 159
sort 8, 62, 69, 99, 201
troff 95, 99–100, 102, 113
wc 158
who 205
command-line arguments 206
command-line variable assignment 206
commas, inserting 54
comment, # 14, 164
comparison expression, value of 181
comparison operator, == 9, 188
> 8
>= 8
comparison operators 181
comparison operators, table of 167
comparison, numeric 168–169, 188
comparison, string 158, 168, 188
compiler model 107
complemented character class 172
compound patterns 169
computation of base-10 logarithm 182
computation of *e* 182
computation of π 182
concatenation in regular expression 173
concatenation operator 156, 182, 186
concatenation, string 12, 27, 75, 156, 158, 182, 186, 189, 199
concordance 98
conditional expression, ?: 52, 180
constant, "..." string 6, 167, 177, 194
constant, numeric 177
constraint graph 144
context-free grammar 87, 120, 122
continue statement 192
continuing long statements 14, 164
control-break program 71, 79, 83, 101
control-flow statements, summary of 190
conversion, % format 184
* format 99
number to string 178, 186
string to number 177, 186
CONVFMT variable 178, 189
Coors 49
cos function 182
cost of array reference 158
countries file 165
cross-reference checking 56
cross-references in manuscripts 95
CSV 33, 38, 68, 164, 166, 195, 203
curl command 31
cycle, graph 145–148, 151
Cygwin xii

data structure, dictionary 193
hash table 193
map 193
successor-list 146
data validation 10, 57
data, name-value 64
regular expressions in 92
self-identifying 64
database attribute 76
database description, relfile 79
database query 73
database table 76
database, multifile 76
database, relational x, 75
date command 33, 205
decrement operator, -- 11, 53, 86, 182
default action 5, 8, 163
default field separator 5, 166
default initialization 11–12, 155, 178, 180, 188, 193–194, 197
delete statement 195
delimiters, balanced 57
dependency description, makefile 149
dependency graph 150
depth-first search algorithm 147, 151
dfs function 148
dictionary data structure 193
divide and conquer xi, 67, 83, 96, 98–99, 104, 135, 159
do statement 192
Dragon book 119
duplicate lines, remove 188
dynamic regular expression 75, 158, 183

e, computation of 182
echo program 206
egrep command 155, 159
else, semicolon before 190–191
emp.data file 1
empty statement 164, 193
end of command-line options, -- 206
end of input, $0 at 12
END pattern 10, 166, 193
END, multiple BEGIN and 143, 166
ENVIRON variable 178
error function 92, 124, 152
error messages, printing 201
error, syntax 4
escape character, \033 173
escape sequence 173, 177
escape sequences, table of 173
evaluation, order of 183
examples, regular expression 174
examples, table of printf 199

executable file 207
exit statement 190, 193
exit status 193, 207
exp function 182
exponentiation operator, ^ 14, 181, 189
expression grammar 120
expression, ?: conditional 52, 180
  assignment 28, 102, 180
  value of comparison 181
  value of logical 181
expressions, field 179
  primary 176
  summary of 179

Farmstead, Hill 49
fflush function 202
field expressions 179
field program 208
field separator, default 5, 166
  input 166, 178, 180, 202
  newline as 61–62, 203
  output 5, 178, 180, 197–199
  regular expression 110, 195, 203
field variables 178
field, $n 5, 178
field, nonexistent 179, 188
fields, named 76, 80
file updating 148
file, /dev/tty 201
  capitals 76
  countries 165
  emp.data 1
  executable 207
  standard error 201
  standard input 202, 208
  standard output 5, 199
FILENAME variable 60, 76, 175, 178
fixed-field input 55
fizzbuzz program 15
floating-point number, regular expression for 174, 183
floating-point precision 177
Floyd, R. W. 137
fmt program 95, 159
FNR variable 175, 178, 204
for ... in statement 194
for statement 15, 192
for(;;) infinite loop 87, 192
forcing coercion to number 187
forcing coercion to string 187
form letters 74
form of Awk program 2, 163
form.gen program 75
form1 program 69
form2 program 70
formal parameters 197
format, program 10, 163, 176, 190, 196
Forth language 116
Fraser, C. W. xiii
FS variable 61, 110, 166, 178, 195, 202
function arguments 197
function definition 163, 196
function with counters, isort 134
function, atan2 182
  check 132
  close 60, 202
  cos 182
  dfs 148
  error 92, 124, 152
  exp 182
  fflush 202
  getline 33, 156, 204
  gsub 25, 54, 75, 93, 97, 156, 185
  heapify 139–140
  hsort 140
  index 55, 184
  int 182
  isort 130
  isplit 34
  length 13
  log 182
  match 124, 156, 178, 184
  max 196
  prefix 78
  qsort 137
  rand 85, 182
  randint 85
  randk 86
  randlet 86
  recursive 55, 89, 136, 197
  sin 182
  split 33–34, 62, 184, 188, 195–196
  sprintf 66, 184
  sqrt 182
  srand 85, 182
  sub 25, 156, 185
  subset 82
  substr 55, 185
  suffix 78
  system 201, 207
  to_csv 39
  unget 79
functions, table of arithmetic 182
  table of string 183
  user-defined 156, 163, 196

Gawk xii, 46, 157–158
gcc command 150
generation, program xi, 59, 96, 142
getline error return 204–205
getline forms, table of 204
getline function 33, 156, 204
getline, side-effects of 204
GitHub xii
global variables 89, 197
Gnuplot 47
Go Awk xii
grammar, arithmetic expression 120
  Awk 122
  context-free 87, 120, 122
grap language 113
graph cycle 145–148, 151
graph language 111
graph, constraint 144
graph, dependency 150
grep command xii, 155, 159, 202
Griswold, R. 161
Grosse, E. H. xiii
gsub function 25, 54, 75, 93, 97, 156, 185
Gusella, R. xiii

happiness, bluebird of 173
hash table 193
hash table data structure 193
hawk calculator 117
headers, records with 63
heapify function 139–140
heapsort algorithm 137
heapsort performance 140
heapsort, profiling 143–144
Herbst, R. T. xiii
hexadecimal escape, \x 173
Hill Farmstead 49
histogram program 53
Hoare, C. A. R. 135
Hoyt, B. xii–xiii
hsort function 140

if-else statement 13, 190
implementation limits 202, 205
in operator 188, 194
increment operator, ++ 11, 134, 181
index function 55, 184
index, KWIC 97
indexing 99
indexing pipeline 104
inf (infinity) 177
infinite loop, for(;;) 87, 192
infix notation 116, 119
info program 74
initialization, default 11–12, 155, 178, 180, 188, 193–194, 197
initializing rand 85
input field separator 166, 178, 180, 202
input line $0 5
input pipe, | 33, 205
input pushback 79, 83
input, fixed-field 55
input, side-effects of 178
inserting commas 54
insertion sort algorithm 129
insertion sort performance 134
int function 182
integer, rounding to nearest 182
interactive test program 133
interactive testing 132
interest program 14
isort function 130
isort function with counters 134
isplit function 34
ix.collapse program 101
ix.format program 104
ix.genkey program 103
ix.rotate program 102
ix.sort1 program 101
ix.sort2 program 103

Java language 193
JavaScript language ix, 193
join command 77
join program 78
join, natural 77
justification, text 72

Kaggle 41
Katakana characters 172
Kernighan, B. W. 113, 117, 123
Kernighan, M. D. xiii
Knuth, Donald Ervin 60
KWIC index 97
kwic program 98

language comparisons, table of 159
language features, new 156
language processor model 107
language, assembly 109
  Forth 116
  grap 113
  graph 111
  Java 193
  JavaScript ix, 193

pattern-directed 112, 114, 127, 132, 155
Perl ix, 156
pic 113
Postscript 116
Python ix, 28, 38, 46–47, 111, 156, 160, 193
q query 75, 80
query 73
REXX 162
SNOBOL4 156, 161
sortgen 113
LaTeX formatter 95, 99
leftmost longest match 185, 203
length function 13
Lesk, M. E. 155
letters, form 74
lex lexical analyzer generator 127, 155
lexical analysis 107, 109
limits, implementation 202, 205
Linderman, J. P. xiii
line continuation after comma 164
linear algorithm 133, 157
linear order 145
lines versus records 163, 203
lines, remove duplicate 188
little languages xi, 107, 132, 134
local variables 89, 156, 196–197
locale 172
locale variable 94
log function 182
logarithm, computation of base-10 182
logical expression, value of 181
logical operators 9, 169, 181
logical operators, precedence of 169
long statements, continuing 14, 164
long string, split 31
ls command 151
Łukasiewicz, Jan, 116

machine dependency 157, 177, 181, 188, 194
make command 148
make program 152
make update algorithm 151
makefile dependency description 149
makeprof program 142
manuscripts, cross-references in 95
map data structure 193
Markdown 95
Martin, R. L. xiii
match function 124, 156, 178, 184
match operator, ~ 167, 170, 174, 181–182
match, leftmost longest 185, 203
matching operators 181
Matplotlib 47, 111
Mawk xii
max function 196
McIlroy, M. D. xiii
metacharacters, regular expression 171
model, language processor 107
Moscovitz, H. S. xiii
multidimensional arrays 81, 89, 156, 195
multifile database 76
multiline records x, 60, 203
multiline string 177
multiple -f options 164
multiple assignment 180
multiple BEGIN and END 143, 166

n log *n* algorithm 137, 140
name-value data 64
named character class 94, 171
named fields 76, 80
names, rules for variable 177
nan (not a number) 177
natural join 77
new language features 156
newline as field separator 61–62, 203
newline character, \n 7, 59, 173
next statement 190, 192
nextfile statement 190, 193
NF variable 5, 13, 178, 204
NF, side-effects on 179, 204
nm command 56
nm.format program 56
nonexistent field 179, 188
nonmatch operator, !~ 167, 170, 174, 181–182
nonterminal symbol 88, 120
NOT operator, ! 9, 169, 181
notation, infix 116, 119
notation, reverse-Polish 116
NR variable 6, 11, 13, 178, 204
null string 12, 89, 167, 185
number or string 186
number to string coercion 130, 156, 167, 186
number to string conversion 178, 186
number, forcing coercion to 187
number, regular expression for floating-point 174, 183
numbers, scientific notation for 177
numeric comparison 168–169, 188
numeric constant 177
numeric subscripts 195
numeric value of a string 188
numeric variables 186

OFMT variable 178, 189
OFS variable 178, 186, 197–198
one-liners 17, 155
operator, ! NOT 9, 169, 181
!~ nonmatch 167, 170, 174, 181–182
% remainder 15, 181, 189
%= assignment 180
&& AND 9, 134, 169, 181
*= assignment 180
++ increment 11, 134, 181
+= assignment 180
-- decrement 11, 53, 86, 182
-= assignment 180
/= assignment 180
= assignment 179
== comparison 9, 188
> comparison 8
>= comparison 8
concatenation 156, 182, 186
in 188, 194
^ exponentiation 14, 181, 189
^= assignment 180
|| OR 9, 169, 181
~ match 167, 170, 174, 181–182
operators, arithmetic 181, 186
assignment 180
associativity of 189
comparison 181
logical 9, 169, 181
matching 181
precedence of 189
precedence of regular expression 173
relational 167, 181
table of arithmetic 189
table of comparison 167
unary 181
option, -- 164
--csv 33, 38, 164, 166, 195, 203, 205
--version xii, 164, 205
-F 164, 203, 205
-f 4, 32, 164, 205, 207
-v 205
OR operator, || 9, 169, 181
order of evaluation 183
ORS variable 61, 178, 197–198
output field separator 5, 178, 180, 197–199
output into pipes 8, 201
output record separator 5, 61, 197–198
output redirection, > 199–200
>> 199–200
| 201
output statements, summary of 197

p12check program 58
Pandas Python package 28, 38
parameter list 89, 196
parameter, array 197
parameter, scalar 197
parameters, formal 197
parenthesis-free notation 116
Parnas, D. L. 98
parser generator, yacc 119, 127, 149–150
parsing, recursive-descent 119, 122
partial order 144
partitioning step, quicksort 136
pattern, BEGIN 10, 166, 206
END 10, 166, 193
range 63, 175
regular expression 169
pattern-action cycle 2, 163
pattern-action statement x, 2, 163, 176, 196
pattern-directed language 112, 114, 127, 132, 155
patterns, compound 169
summary of 165
summary of string-matching 169
percent program 53
performance measurements, table of 158
performance, heapsort 140
insertion sort 134
quicksort 137
Perl language ix, 156
permuted index 98
pic language 113
Pike, R. 117
pipe, | input 33, 205
pipeline, indexing 104
pipes, output into 8, 201
Poage, J. 161
Polish notation 116
Polonsky, I. 161
POSIX standard xii
Postscript language 116
pr command 150
precedence of logical operators 169
precedence of operators 189
precedence of regular expression operators 173

precision, floating-point 177
predecessor node 145
prefix function 78
prep1 program 68
prep2 program 70
primary expressions 176
print >>*file* 197
print >*file* 69, 197
print statement 5, 197
print |*file* 197
printf examples, table of 199
printf specifications, table of 199
printf statement 7, 72, 166, 199
printf, %% in 59
printing a blank line 10, 198
printing error messages 201
printprof program 142
priority queue 137
processor, #include 205, 207
profiling 142
profiling heapsort 143–144
program format 10, 163, 176, 190, 196
program generation xi, 59, 96, 142
program, addcomma 54
  addup 28
  addup2 52
  addup3 52
  arith 91
  asm 110
  awk.parser 124
  batch sort test 131
  bmi 21
  bundle 60
  calc1 116
  calc2 117
  calc3 121
  cf 22
  charfreq 158
  check1 65
  check2 65
  check3 66
  checkgen 59
  checkpasswd 58
  cliche 87
  colcheck 57
  echo 206
  field 208
  fizzbuzz 15
  fmt 95, 159
  form.gen 75
  form1 69
  form2 70
  histogram 53
  info 74
  interest 14
  ix.collapse 101
  ix.format 104
  ix.genkey 103
  ix.rotate 102
  ix.sort1 101
  ix.sort2 103
  join 78
  kwic 98
  make 152
  makeprof 142
  nm.format 56
  p12check 58
  percent 53
  prep1 68
  prep2 70
  printprof 142
  qawk 82
  quiz 92
  quote 31
  randline 86
  rtsort 148
  sentgen 89
  seq 206
  sortgen 114
  sumcomma 54
  table 72
  test framework 135
  tsort 146
  unbundle 60
  word count 13, 92
  wordfreq 94
  xref 97
prompt character 2
prototyping x–xi, 58, 127, 161
pseudo-code xi, 129
ptx command 98
pushback, input 79, 83
Python language ix, 28, 38, 46–47, 111, 156, 160, 193
Python package, BeautifulSoup 31
Python package, Pandas 28, 38

q query language 75, 80
qawk program 82
qawk query processor 81
qsort function 137
quadratic algorithm 133, 137, 157
query language 73
queue 145
queue, priority 137
quicksort algorithm 135
quicksort partitioning step 136
quicksort performance 137
quiz program 92
quote program 31
quotes, '' 2, 4, 207
quoting in regular expressions 172, 174, 183, 185

Ramming, J. C. 123
rand function 85, 182
rand, initializing 85
randint function 85
randk function 86
randlet function 86
randline program 86
random permutation algorithm 87
random selection algorithm 86
random sentences 87
range pattern 63, 175
RateBeer 41
record separator, output 5, 61, 197–198
record variable, $0 5, 178
records with headers 63
records, lines versus 163, 203
records, multiline x, 60, 203
recursive function 55, 89, 136, 197
recursive-descent parsing 119, 122
redirection, > output 199–200
  >> output 199–200
  | output 201
regular expression character class 172
regular expression examples 174
regular expression field separator 110, 195, 203
regular expression for floating-point number 174, 183
regular expression metacharacters 171
regular expression operators, precedence of 173
regular expression pattern 169
regular expression, $ 92, 172
  () 172
  * 173
  + 173
  . 172
  ? 173
  concatenation in 173
  dynamic 75, 158, 183
  RS as 204
  [^...] 172
  ^ 92, 172
  {} 173
  | 169, 172
regular expressions in data 92
regular expressions, quoting in 172, 174, 183, 185
  strings as 182
  summary of 171
  table of 174
relation, universal 80
relational database x, 75
relational operators 167, 181
relfile database description 79
remainder operator, % 15, 181, 189
remove duplicate lines 188
REPL 119
report generation 67
return statement 196
reverse input line order 193
reverse program 16
reverse-Polish notation 116
REXX language 162
RLENGTH variable 178, 184
Robbins, A. D. xii–xiii
Rochkind, Marc 59
rounding to nearest integer 182
RS as regular expression 204
RS variable 61–62, 178, 204
RSTART variable 178, 184
rtsort program 148
rules for variable names 177
running an Awk program 3
running time of Awk programs 158–159

scaffolding 129, 132, 153
scalar parameter 197
Schmitt, G. xiii
scientific notation for numbers 177
sed command xii, 155, 159
self-identifying data 64
semicolon 10, 163, 176, 190, 196
semicolon as empty statement 193
semicolon before else 190–191
sentence generation 88
sentences, random 87
sentgen program 89
separator, blank line 61
  default field 5, 166
  input field 166, 178, 180, 202
  output field 5, 178, 180, 197–199
  output record 5, 61, 197–198
seq program 206
Sethi, R. 119

shell command interpreter 4, 207
shell script 23, 162
side-effects of assignment 186
side-effects of getline 204
side-effects of input 178
side-effects of sub 185
side-effects of test 188, 195
side-effects on $0 179, 186
side-effects on NF 179, 204
sin function 182
Sites, R. xiii
SNOBOL4 language 156, 161
sort command 8, 62, 69, 99, 201
sort key 70, 94, 102, 113
sort options 69, 99, 101, 114
sort options, avoiding 69, 94, 113
sort programs, testing 131
sort test program, batch 131
sortgen language 113
sortgen program 114
sorting address list 62
sorting, topological 144
split function 33–34, 62, 184, 188, 195–196
split long string 31
sprintf function 66, 184
sqrt function 182
srand function 85, 182
stack 116
standard error file 201
standard input file 202, 208
standard input filename, - 91, 164, 204–205
standard output file 5, 199
statement, break 192
  continue 192
  delete 195
  do 192
  empty 164, 193
  exit 190, 193
  for 15, 192
  for ... in 194
  if-else 13, 190
  next 190, 192
  nextfile 190, 193
  pattern-action x, 2, 163, 176, 196
  print 5, 197
  printf 7, 72, 166, 199
  return 196
  while 14, 191
statements, continuing long 14, 164
  summary of control-flow 190
  summary of output 197
status return 193, 207
string comparison 158, 168, 188
string concatenation 12, 27, 75, 156, 158, 182, 186, 189, 199
string constant, "..." 6, 167, 177, 194
string functions, table of 183
string or number 186
string to number coercion 156, 167, 186
string to number conversion 177, 186
string variables 12, 186
string, forcing coercion to 187
  multiline 177
  null 12, 89, 167, 185
  numeric value of a 188
  split long 31
string-matching patterns, summary of 169
strings as regular expressions 182
sub function 25, 156, 185
sub, side-effects of 185
subscripts, array 193–195
subscripts, numeric 195
SUBSEP variable 178, 196
subset function 82
substitution, & in 55, 185
substr function 55, 185
successor node 145
successor-list data structure 146
suffix function 78
sumcomma program 54
summary of actions 176
summary of control-flow statements 190
summary of expressions 179
summary of output statements 197
summary of patterns 165
summary of regular expressions 171
summary of string-matching patterns 169
summing columns 51
Swartwout, D. xiii
symbol table 107, 110, 127
syntax error 4
system function 201, 207

tab character, \t 14, 166, 173
table of arithmetic functions 182
table of arithmetic operators 189
table of assembler instructions 108
table of built-in variables 178
table of comparison operators 167
table of escape sequences 173
table of getline forms 204
table of language comparisons 159
table of performance measurements 158
table of printf examples 199
table of printf specifications 199
table of regular expressions 174
table of string functions 183
table program 72
table, symbol 107, 110, 127
tables, base and derived 79
terminal symbol 88, 120
test framework program 135
test program, interactive 133
test, side-effects of 188, 195
testing sort programs 131
testing, boundary condition 131
testing, interactive 132
text justification 72
timing tests 157
to_csv function 39
topological sort algorithm 145
topological sorting 144
translator model 107
tree, binary 138
Trickey, H. W. xiii
troff command 95, 99–100, 102, 113
tsort program 146
Tukey, J. W. 35, 47, 49

Ullman, J. D. 119
unary operators 181
unbundle program 60
underscore, _ 177
unget function 79
Unicode xii, 41, 45, 172, 174
Unicode escape, \u 173
uninitialized variables 194, 200
universal relation 80
update algorithm, make 151
updating, file 148
user-defined functions 156, 163, 196
UTF-8 xii, 41, 167, 183

value of a string, numeric 188
value of comparison expression 181
value of logical expression 181
van Eijk, P. xiii
Van Wyk, C. J. xiii
variable assignment, command-line 206
variable names, rules for 177
variable, $0 record 5, 178
  ARGC 23, 178, 206
  ARGV 22–23, 91, 178, 206–208
  CONVFMT 178, 189
  ENVIRON 178
  FILENAME 60, 76, 175, 178
  FNR 175, 178, 204
  FS 61, 110, 166, 178, 195, 202
  locale 94
  NF 5, 13, 178, 204
  NR 6, 11, 13, 178, 204
  OFMT 178, 189
  OFS 178, 186, 197–198
  ORS 61, 178, 197–198
  RLENGTH 178, 184
  RS 61–62, 178, 204
  RSTART 178, 184
  SUBSEP 178, 196
variables, field 178
  global 89, 197
  local 89, 156, 196–197
  numeric 186
  string 12, 186
  table of built-in 178
  uninitialized 194, 200
versions, Awk xii, 38, 157, 164, 205

wc command 158
while statement 14, 191
who command 205
wild-card characters 169
Williams, J. W. J. 137
Windows Subsystem for Linux (WSL) xii
word count program 13, 92
wordfreq program 94
www.awk.dev xii

xref program 97

yacc parser generator 119, 127, 149–150
Yannakakis, M. xiii
Yigit, O. xiii

Zakharov, D. xii