The AWK Programming Language

Second Edition

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







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.







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.

xiii

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

1

An Awk Tutorial

Awk is a convenient and expressive programming language that can be applied to a wide variety of computing and data-manipulation tasks. This chapter is a tutorial, designed to let you start writing your own programs as quickly as possible. The remaining chapters show how Awk can be used to solve problems from many different areas; the reference manual in Appendix A describes the whole language in detail. Throughout the book, we have tried to pick examples that you might find useful, interesting, and instructive.

1.1 Getting Started

Useful Awk programs are often short, just a line or two. Suppose you have a file called emp.data that contains three fields of information about your employees: name, pay rate in dollars per hour, and number of hours worked, one employee record per line with the fields separated by spaces or tabs, like this:

Beth	21	0
Dan	19	0
Kathy	15.50	10
Mark	25	20
Mary	22.50	22
Susie	17	18

Now you want to print the name and pay (rate times hours) for everyone who worked more than zero hours. This is the kind of job that Awk is meant for, so it's easy. Type this command line (after the command-line prompt \$):

\$ awk '\$3 > 0 { print \$1, \$2 * \$3 }' emp.data

You should get this output:

```
Kathy 155
Mark 500
Mary 495
Susie 306
```

This command line tells the system to run Awk, using the program inside the quote

characters, taking its data from the input file emp.data. The part inside the quotes is the complete Awk program. It consists of a single *pattern-action statement*. The pattern \$3 > 0 matches every input line in which the third column, or *field*, is greater than zero, and the action

{ print \$1, \$2 * \$3 }

prints the first field and the product of the second and third fields of each matched line.

To print the names of those employees who did not work, type this command line:

\$ awk '\$3 == 0 { print \$1 }' emp.data

In this example, the pattern \$3 == 0 matches each line in which the third field is equal to zero, and the action

```
{ print $1 }
```

prints its first field.

As you read this book, try running and modifying the programs that are presented. Since most of the programs are short, you'll quickly get an understanding of how Awk works. On a Unix system, the two transactions above would look like this in a terminal window:

```
$ awk '$3 > 0 { print $1, $2 * $3 }' emp.data
Kathy 155
Mark 500
Mary 495
Susie 306
$ awk '$3 == 0 { print $1 }' emp.data
Beth
Dan
$
```

The \$ at the beginning of such lines is the prompt from the system; it may be different on your computer.

The Structure of an Awk Program

Let's step back for a moment and look at what's going on. In the command lines above, the parts between the quote characters are programs written in the Awk programming language. An Awk program is a sequence of one or more pattern-action statements:

The basic operation of Awk is to scan a sequence of input lines, from any number of files, one after another, searching for lines that are *matched* by any of the patterns in the program. The precise meaning of "match" depends on the pattern in question; for patterns like \$3 > 0, it means "the condition is true."

Every input line is tested against each of the patterns in turn. For each pattern that matches, the corresponding action (which may involve multiple steps) is performed. Then the next line is read and the matching starts over. This continues until all the input lines have been read.

The programs above are typical examples of patterns and actions. Here's an example of a single pattern-action statement; for every line in which the third field is zero, the first field is

printed.

\$3 == 0 { print \$1 }

It prints

Beth Dan

Either the pattern or the action (but not both) in a pattern-action statement may be omitted. If a pattern has no action, for example,

\$3 == 0

then each line that the pattern matches (that is, each line for which the condition is true) is printed. This program prints the two lines from the emp.data file where the third field is zero:

Beth 21 0 Dan 19 0

If there is an action with no pattern, for example,

{ print \$1 }

then the action, in this case printing the first field, is performed for every input line.

Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns. Completely blank lines are ignored.

Running an Awk Program

There are two ways to run an Awk program. You can type a command line of the form

awk 'program' input files

to run the program on each of the specified input files. For example, you could type

awk '\$3 == 0 { print \$1 }' file1 file2

to print the first field of every line of file1 and then file2 in which the third field is zero. You can omit the input files from the command line and just type

awk 'program'

In this case Awk will apply the *program* to whatever you type next on your terminal until you type an end-of-file signal (Control-D on Unix systems). Here is a sample of a session on Unix; the italic text is what the user typed.

```
$ awk '$3 == 0 { print $1 }'
Beth
     21
          0
Beth
Dan
     19 0
Dan
Kathy 15.50
             10
Kathy 15.50
             0
Kathy
     22.50
Mary
           22
. . .
```

This behavior makes it easy to experiment with Awk: type your program, then type data at it and see what happens. We encourage you to try the examples and variations on them.

Notice that the program is enclosed in single quotes on the command line. This protects characters like \$ in the program from being interpreted by the shell and also allows the program to be longer than one line.

This arrangement is convenient when the program is only one or two lines long. If the program is longer, however, it may be easier to put it into a separate file, say progfile, and type the command line

awk -f progfile optional list of input files

The -f option instructs Awk to fetch the program from the named file. Any filename can be used in place of progfile; if the filename is -, the standard input is used.

Errors

If you make an error in an Awk program, Awk will give you a diagnostic message. For example, if you type a bracket when you meant to type a brace, like this:

\$ awk '\$3 == 0 [print \$1 }' emp.data

you will get a message like this:

```
awk: syntax error at source line 1
context is
    $3 == 0 >>> [ <<<
    extra }
    missing ]
awk: bailing out at source line 1</pre>
```

"Syntax error" means that you have made a grammatical error that was detected at the place marked by >>> <<<. "Bailing out" means that no recovery was attempted. Sometimes you get a little more help about what the error was, such as a report of mismatched braces or parentheses.

Because of the syntax error, Awk did not try to execute this program. Some errors, however, may not be detected until your program is running. For example, if you attempt to divide a number by zero, Awk will stop its processing and report the input line and the line number in the program at which the division was attempted.

1.2 Simple Output

The rest of this chapter contains a collection of short, typical Awk programs based on manipulation of the emp.data file above. We'll explain briefly what's going on, but these examples are meant mainly to suggest useful operations that are easy to do with Awk — printing fields, selecting input, and transforming data. We are not showing everything that Awk can do by any means, nor are we going into many details about the specific things presented here. But by the end of this chapter, you will be able to accomplish quite a bit, and you'll find it much easier to read the later chapters.

We will usually show just the program, not the whole command line. In every case, the program can be run either by enclosing it in quotes as the first argument of the awk command, as above, or by putting it in a file and invoking Awk on that file with the -f option.

There are only two types of data in Awk: numbers and strings of characters. The emp.data file is typical of this kind of information — a sequence of text lines containing a mixture of words and numbers separated by spaces and/or tabs.

Awk reads its input one line at a time and splits each line into fields, where, by default, a field is a sequence of characters that doesn't contain any spaces or tabs. The first field in the current input line is called \$1, the second \$2, and so forth. The entire line is called \$0. The number of fields can vary from line to line.

Often, all we need to do is print some or all of the fields of each line, perhaps performing some calculations. The programs in this section are all of that form.

Printing Every Line

If an action has no pattern, the action is performed for all input lines. The statement print by itself prints the current input line, so the program

```
{ print }
```

prints all of its input on the standard output. Since \$0 is the whole line,

```
{ print $0 }
```

does the same thing.

Printing Specific Fields

More than one item can be printed on the same output line with a single print statement. The program to print the first and third fields of each input line is

```
{ print $1, $3 }
```

With emp.data as input, it produces

```
Beth 0
Dan 0
Kathy 10
Mark 20
Mary 22
Susie 18
```

Expressions separated by a comma in a print statement are, by default, separated by a single space when they are printed. Each line produced by print ends with a newline character. Both of these defaults can be changed; we'll show how in later examples and in Section A.4.2 of the reference manual.

NF, the Number of Fields

It might appear that you must always refer to fields as \$1, \$2, and so on, but any expression can be used after \$ to denote a field number; the expression is evaluated and its numeric value is used as the field number. Awk counts the number of fields in the current input line and stores the count in a built-in variable called NF. Thus, the program

```
{ print NF, $1, $NF }
```

prints the number of fields and the first and last fields of each input line.

Computing and Printing

You can also do computations on the field values and include the results in what is printed. The program

{ print \$1, \$2 * \$3 }

is a typical example. It prints the name and total pay (that is, rate times hours) for each employee:

Beth 0 Dan 0 Kathy 155 Mark 500 Mary 495 Susie 306

In a moment we'll show how to make this output look better.

Printing Line Numbers

Awk provides another built-in variable, called NR, that counts the number of lines (records) read so far. We can use NR and \$0 to prefix each line of emp.data with its line number, like this:

{ print NR, \$0 }

The output looks like this:

1	Beth	21	0
2	Dan	19	0
3	Kathy	15.50	10
4	Mark	25	20
5	Mary	22.50	22
6	Susie	17	18

Putting Text in the Output

You can also print words in the midst of fields and computed values, by including quoted strings of characters in the list:

{ print "total pay for", \$1, "is", \$2 * \$3 }

prints

```
total pay for Beth is 0
total pay for Dan is 0
total pay for Kathy is 155
total pay for Mark is 500
total pay for Mary is 495
total pay for Susie is 306
```

In the print statement, the text inside the double quotes is printed along with the fields and computed values.

1.3 Formatted Output

The print statement is meant for quick and easy output. To format the output exactly the way you want it, you may have to use the printf statement. As we will see in many subsequent examples, printf can produce almost any kind of output, but in this section we'll only show a few of its capabilities. Section A.4.3 gives the details.

Lining Up Fields

The printf statement has the form

printf(format, value1, value2, ..., valuen)

where *format* is a string that contains text to be printed verbatim, interspersed with specifications of how each of the values is to be printed. A specification is a followed by a few characters that control the format of a *value*. The first specification tells how *value*₁ is to be printed, the second how *value*₂ is to be printed, and so on. Thus, there must be as many specifications in *format* as *values* to be printed. (The printf statement is almost the same as the printf function in the standard C library.)

Here's a program that uses printf to print the total pay for every employee:

{ printf("total pay for %s is \$%.2f\n", \$1, \$2 * \$3) }

The specification string in the printf statement contains two \$ specifications. The first, \$s, says to print the first value, \$1, as a string of characters; the second, \$.2f, says to print the second value, \$2 * \$3, as a number with 2 digits after the decimal point. Everything else in the specification string, including the dollar sign, is printed verbatim; the \n at the end of the string stands for a newline, which causes subsequent output to begin on the next line. With emp.data as input, this program yields:

```
total pay for Beth is $0.00
total pay for Dan is $0.00
total pay for Kathy is $155.00
total pay for Mark is $500.00
total pay for Mary is $495.00
total pay for Susie is $306.00
```

With printf, no spaces or newlines are produced automatically; you must create them yourself. If you forget the n, the output will all be on a single line.

Here's another program that prints each employee's name and pay:

{ printf("%-8s \$%6.2f\n", \$1, \$2 * \$3) }

The first specification, -8s, prints a name as a string of characters left-justified in a field 8 characters wide; the minus sign signals that the name is to be left-justified in its field. The second specification, 6.2f, prints the pay as a number with two digits after the decimal point, in a field 6 characters wide:

Beth	\$	0.00
Dan	\$	0.00
Kathy	\$15	5.00
Mark	\$50	00.00
Mary	\$49	95.00
Susie	\$30	06.00

We'll show lots more examples of printf as we go along; the full story can be found in Section A.4.3.

Sorting the Output

Suppose you want to print all the data for each employee, along with his or her pay, sorted in order of increasing pay. The easiest way is to use Awk to prefix the total pay to each employee record, and run that output through a sorting program.

On Unix, the command line

```
awk '{ printf("%6.2f %s\n", $2 * $3, $0) }' emp.data | sort
```

pipes the output of Awk into the sort command, and produces:

0.00	Beth	21	0
0.00	Dan	19	0
155.00	Kathy	15.50	10
306.00	Susie	17	18
495.00	Mary	22.50	22
500.00	Mark	25	20

It's quite possible to write an efficient sort program in Awk itself; there's an example Quicksort in Chapter 8. But most of the time, it's more productive to use existing tools like sort.

1.4 Selection

Awk patterns are good for selecting interesting lines from the input for further processing. Since a pattern without an action prints all lines matching the pattern, many Awk programs consist of nothing more than a single pattern. This section gives some examples of useful patterns.

Selection by Comparison

This program uses a comparison pattern to select the records of employees who earn \$20 or more per hour, that is, lines in which the second field is greater than or equal to 20:

\$2 >= 20

It selects these lines from emp.data:

Beth	21	0
Mark	25	20
Mary	22.50	22

Selection by Computation

The program

\$2 * \$3 > 200 { printf("\$%.2f for %s\n", \$2 * \$3, \$1) }

prints the pay of those employees whose total pay exceeds \$200:

\$500.00 for Mark \$495.00 for Mary \$306.00 for Susie

Selection by Text Content

Besides numeric tests, you can select input lines that contain specific words or phrases. This program prints all lines in which the first field is Susie:

\$1 == "Susie"

The operator == tests for equality. You can also look for text containing any of a set of letters, words, and phrases by using patterns called *regular expressions*. This program prints all lines that contain Susie anywhere:

/Susie/

The output is this line:

Susie 17 18

Regular expressions can be used to specify much more elaborate text patterns; Section A.1.4 contains a full discussion.

Combinations of Patterns

Patterns can be combined with parentheses and the logical operators &&, ||, and !, which stand for AND, OR, and NOT. The program

\$2 >= 20 || \$3 >= 20

prints those lines where \$2 is at least 20 or \$3 is at least 20:

Beth	21	0
Mark	25	20
Mary	22.50	22

Lines that satisfy both conditions are printed only once. Contrast this with the following program, which consists of two patterns:

\$2 >= 20 \$3 >= 20

This program prints an input line twice if it satisfies both conditions:

Beth	21	0
Mark	25	20
Mark	25	20
Mary	22.50	22
Mary	22.50	22

Note that the program

!(\$2 < 20 && \$3 < 20)

prints lines where it is *not* true that \$2 is less than 14 *and* \$3 is less than 20; this condition is equivalent to the first one above, though less readable.

Data Validation

Real data always contains errors. Awk is an excellent tool for checking that data is in the right format and has reasonable values, a task called *data validation*.

Data validation is essentially negative: instead of printing lines with desirable properties, one prints lines that are suspicious. The following program uses comparison patterns to apply five plausibility tests to each line of emp.data:

```
NF != 3 { print $0, "number of fields is not equal to 3" }
$2 < 15 { print $0, "rate is too low" }
$2 > 25 { print $0, "rate exceeds $25 per hour" }
$3 < 0 { print $0, "negative hours worked" }
$3 > 60 { print $0, "too many hours worked" }
```

If there are no errors, there's no output.

BEGIN and END

The special pattern BEGIN matches before the first line of the first input file is read, and END matches after the last line of the last file has been processed. This program uses BEGIN to print a heading; the words are separated by the right number of spaces.

BEGIN { print "NAME RATE HOURS"; print "" }
 { print }

The output is:

NAME	RATE	HOURS
Beth	21	0
Dan	19	0
Kathy	15.50	10
Mark	25	20
Mary	22.50	22
Susie	17	18

You can put several statements on a single line if you separate them by semicolons. Notice that print "" prints a blank line, quite different from just plain print, which prints the current input line.

1.5 Computing with Awk

An action is a sequence of statements separated by newlines or semicolons. You have already seen examples in which the action was a single print or printf statement. This section provides examples of statements for performing simple numeric and string computations. In these statements you can use not only the built-in variables like NF, but you can create your own variables for performing calculations, storing data, and the like. In Awk, usercreated variables are not declared; they come into existence when you use them.

Counting

This program uses a variable emp to count the number of employees who have worked more than 15 hours:

\$3 > 15 { emp = emp + 1 }
END { print emp, "employees worked more than 15 hours" }

For every line in which the third field exceeds 15, the previous value of emp is incremented by 1. With emp.data as input, this program yields:

3 employees worked more than 15 hours

Awk variables used as numbers begin life with the value 0, so we didn't need to initialize the variable emp.

Statements like

emp = emp + 1

occur so frequently that C and languages inspired by it provide an increment operator ++ as a shorthand equivalent:

emp++

There is a corresponding decrement operator -- that will appear shortly.

We can rewrite the counting example with ++, like this:

```
$3 > 15 { emp++ }
END { print emp, "employees worked more than 15 hours" }
```

Computing Sums and Averages

To count the number of employees, we could instead use the built-in variable NR, which holds the number of lines read so far; its value at the end of all input is the total number of lines read.

```
END { print NR, "employees" }
```

The output is:

```
6 employees
```

Here is a program that uses NR to compute the average pay:

```
{ pay = pay + $2 * $3 }
END { print NR, "employees"
    print "total pay is", pay
    print "average pay is", pay/NR
}
```

The first action accumulates the total pay for all employees. The END action prints

6 employees total pay is 1456 average pay is 242.667

Clearly, printf could be used to produce neater output, for example to produce exactly two digits after the decimal point. There's also a potential error: in the unlikely case that NR is zero, the program will attempt to divide by zero and thus will generate an error message.

The operator += is an abbreviation for incrementing a variable: it increments the variable on its left by the value of the expression on its right, so the first line of the program above could be more compactly written as { pay += \$2 * \$3 }

Handling Text

Awk variables can hold strings of characters as well as numbers. This program finds the employee who is paid the most per hour:

```
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print "highest hourly rate:", maxrate, "for", maxemp }
```

It prints

highest hourly rate: 25 for Mark

In this program the variable maxrate holds a numeric value, while the variable maxemp holds a string. If there are several employees who all make the same maximum pay, this program reports only the first.

String Concatenation

New strings may be created by pasting together old ones; this operation is called *concate-nation*. The string concatenation operation is represented in an Awk program by writing string values one after the other; there is no explicit concatenation operator. (In hindsight, this design might not be ideal, because it can sometimes lead to hard-to-spot errors.)

As an illustration of string concatenation, the program

```
{ names = names $1 " " }
END { print names }
```

collects all the employee names into a single string, by appending each name and a space to the previous value in the variable names. The value of names is printed by the END action:

```
Beth Dan Kathy Mark Mary Susie
```

At every input line, the first statement in the program concatenates three strings: the previous value of names, the first field, and a space; it then assigns the resulting string to names. Thus, after all input lines have been read, names contains a single string consisting of the names of all the employees, each followed by a space (so there is an invisible space at the end of the string). Variables used to store strings begin life holding the null string (that is, the string containing no characters), so in this program names did not need to be explicitly initialized.

Printing the Last Input Line

Built-in variables like NR retain their value in an END action, and so do fields like \$0. The program

```
END { print $0 }
```

is one way to print the last input line.

Susie 17 18

Built-in Functions

We have already seen that Awk provides built-in variables that maintain frequently used quantities like the number of fields and the input line number. Similarly, there are built-in functions for computing other useful values.

Besides arithmetic functions for square roots, logarithms, random numbers, and the like, there are also functions that manipulate text. One of these is length, which counts the number of characters in a string. For example, this program computes the length of each person's name:

```
{ print $1, length($1) }
```

The result:

Beth 4 Dan 3 Kathy 5 Mark 4 Mary 4 Susie 5

Counting Lines, Words, and Characters

This program uses length, NF, and NR to count the number of lines, words, and characters in the input, like the Unix program wc. For convenience, we'll treat each field as a word, though that is a bit of a simplification.

```
{ nc += length($0) + 1
    nw += NF
}
END { print NR, "lines,", nw, "words,", nc, "characters" }
```

The file emp.data has

6 lines, 18 words, 71 characters

We have added 1 to nc to count the newline character at the end of each input line, because \$0 doesn't include it.

1.6 Control-Flow Statements

Awk provides an if-else statement for making decisions and several statements for writing loops, all modeled on those found in the C programming language. They can only be used in actions.

If-Else Statement

The following program computes the total and average pay of employees making more than \$30 an hour. It uses an *if* to defend against any potential division by zero in computing the average pay.

The output for emp.data is:

No employees are paid more than \$30/hour

In the if-else statement, the condition following the if is evaluated. If it is true, the first print statement is performed. Otherwise, the second print statement is performed. Note that we can continue a long statement over several lines by breaking it after a comma.

Note also that if an i f statement controls only a single statement, no braces are necessary, though they are if more than one statement is controlled. This version

```
$2 > 30 { n++; pay += $2 * $3 }
END { if (n > 0) {
        print n, "employees, total pay is", pay,
            " average pay is", pay/n
        } else {
            print "No employees are paid more than $30/hour"
        }
    }
```

uses braces around both if and else parts to make it clear what the scope of control is. In general, it's good practice to use such redundant braces.

While Statement

A while statement has a condition and a body. The statements in the body are performed repeatedly while the condition is true. This program shows how the value of an amount of money invested at a particular interest rate grows over a number of years, using the formula *value* = *amount* $(1 + rate)^{years}$.

```
# interest1 - compute compound interest
# input: amount rate years
# output: compounded value at the end of each year
{ i = 1
   while (i <= $3) {
      printf("\t%.2f\n", $1 * (1 + $2) ^ i)
      i++
   }
}</pre>
```

The condition is the parenthesized expression after the while; the loop body consists of the two statements enclosed in braces after the condition. The \t in the printf specification string stands for a tab character; the $^$ is the exponentiation operator. Text from a # to the end of the line is a *comment*, which is ignored by Awk but should be helpful to readers of the program who want to understand what is going on.

You can type triplets of numbers at this program to see what various amounts, rates, and years produce. For example, this transaction shows how \$1,000 grows at 5% and 10% compound interest for five years; user input is shown in *this font*:

For Statement

Another statement, for, compresses into a single line the initialization, test, and increment that are part of most loops; it is also copied from C. Here is the previous interest computation with a for:

The initialization i = 1 is performed once. Next, the condition $i \le 3$ is tested; if it is true, the body of the loop, which is a single printf statement, is performed. Then the increment i++ is performed after the body, and the next iteration of the loop begins with another test of the condition. The code is more compact, and because the body of the loop is only a single statement, no braces are needed to enclose it.

This example of a for statement is the standard idiomatic way to express a loop that goes from 1 to some upper limit inclusive. If you see a loop with a different initialization or termination, take a second look to be sure it's doing the right thing.

FizzBuzz

As a fun example of loops and conditionals, here's an implementation of FizzBuzz, which is sometimes used as a minimal check on whether a job applicant can program at all. The task is to write a program that prints the numbers from 1 to 100, but if the number is divisible by 3, it prints "fizz"; if divisible by 5 it prints "buzz"; and if divisible by both it prints "fizzbuzz."

The program uses the modulus or remainder operator %, which produces the remainder of a division.

```
awk '
BEGIN {
  for (i = 1; i <= 100; i++) {
    if (i%15 == 0) # divisible by both 3 and 5
      print i, "fizzbuzz"
    else if (i%5 == 0)
      print i, "buzz"
    else if (i%3 == 0)
      print i, "fizz"
    else
      print i
  }
}</pre>
```

All of the computation is done in the BEGIN block; any filename arguments are simply ignored. Note how the else ifs are at the same level of indentation, to make it clear that this is a sequence of decisions.

1.7 Arrays

Awk provides arrays for storing groups of related values. This program prints its input in reverse order by line. The first action stores input lines in successive elements of the array line; that is, the first line goes into line[1], the second line into line[2], and so on. The END action uses a while statement to print the lines from the array from last to first:

With emp.data, the output is

Susie 17 18 22.50 22 Marv 25 20 Mark 15.50 10 Kathv Dan 19 0 Beth 21 0

Here is the same example with a for statement:

The subscripts in this example are numeric, but one of the most useful features of Awk is that array subscripts are not limited to numeric values; they can be arbitrary strings of characters. We will illustrate such subscripts in later chapters.

1.8 Useful One-liners

Although Awk can be used to write programs of some complexity, many useful programs are no more complicated than what we've seen so far. Here is a collection of short programs that you might find handy and/or instructive. Most of them are variations on material we have already covered.

Print the total number of input lines:

END { print NR }

Print the first 10 input lines:

```
NR <= 10
```

Print the tenth input line:

NR == 10

Print every tenth input line, starting with line 1:

NR % 10 == 1

Print the last field of every input line:

{ print \$NF }

Print the last field of the last input line:

END { print \$NF }

Print every input line with more than four fields:

NF > 4

Print every input line that does not have exactly four fields:

NF ! = 4

Print every input line in which the last field is greater than 4:

\$NF > 4

Print the total number of fields in all input lines:

{ nf += NF } END { print nf }

Print the total number of lines that contain Beth:

```
/Beth/ { nlines++ }
END { print nlines }
```

Print the largest first field and the line that contains it (assumes some \$1 is positive):

```
$1 > max { max = $1; maxline = $0 }
END { print max, maxline }
```

Print every line that has at least one field (that is, not empty or all spaces):

NF > 0

Print every line longer than 80 characters:

length(\$0) > 80

Print the number of fields in every line followed by the line itself:

```
{ print NF, $0 }
```

Print the first two fields, in opposite order, of every line:

{ print \$2, \$1 }

Interchange the first two fields of every line and then print the line:

{ temp = \$1; \$1 = \$2; \$2 = temp; print }

Print every line preceded by its line number:

{ print NR, \$0 }

Print every line with the first field replaced by the line number:

{ \$1 = NR; print }

Print every line after erasing the second field:

{ \$2 = ""; print }

Print in reverse order the fields of every line:

```
{ for (i = NF; i > 0; i--) printf("%s ", $i)
    printf("\n")
}
```

Print the sums of the fields of every line:

```
{ sum = 0
  for (i = 1; i <= NF; i++) sum = sum + $i
  print sum
}</pre>
```

Add up all fields in all lines and print the sum:

{ for (i = 1; i <= NF; i++) sum = sum + \$i }
END { print sum }</pre>

Print every line after replacing each field by its absolute value:

```
{ for (i = 1; i <= NF; i++) if ($i < 0) $i = -$i
print
}</pre>
```

1.9 What Next?

You have now seen the essentials of Awk. An Awk program is a sequence of patternaction statements. Awk tests every input line against the patterns in order, and when a pattern matches, performs the corresponding action. Patterns can involve numeric and string comparisons, and actions can include computation and formatted printing. Besides reading through the input files automatically, Awk splits each input line into fields. It also provides a number of built-in variables and functions, and lets you define your own as well. With this combination of features, many useful computations can be expressed by short programs, because the details that would be needed in another language are handled implicitly in an Awk program.

The rest of the book elaborates on these basic ideas. We encourage you to begin writing programs as soon as possible. This will give you familiarity with the language and make it easier to understand larger programs. Furthermore, nothing answers questions so well as some simple experiments. You should also browse through the whole book; each example conveys something about the language, either about how to use a particular feature, or how to create an interesting program.

Language features are introduced as needed in the examples, but are often not fully specified; the reference manual in Appendix A has all the details, illustrated with more examples, so you might find it useful to move back and forth between the next chapters and the manual. This page intentionally left blank

2

Awk in Action

Awk is useful for creating small tools and personal scripts that help you to automate repetitive tasks or to deal with some weirdly specific computation that you care about but no one else does.

In this chapter we're going to look at a number of examples. These specific programs may not be directly useful to you (though with luck a few will be), but they might give you ideas about programs that you could write yourself, or techniques that you could apply in your own programming.

The chapter is organized around simple computations, and the basic operations of selection, transformation, and summarization that Awk was originally designed for. Each of the examples is meant to be of intrinsic interest or utility, to show something about Awk itself, and to illustrate some useful programming technique. Some involve not just Awk but also rely on other standard Unix tools, and effective use of the Unix environment.

2.1 Personal Computation

Awk is a good programmable calculator, with which you can encapsulate a computation in a command that becomes part of your personal set of tools. This section illustrates several tiny ones that we have found useful.

Body Mass Index

Body mass index or BMI is a widely-used measure of body fat that computes a single number from height and weight with the formula $bmi = weight / height^2$. "Normal" BMI is between 18 and 25, from 25 to 30 is "overweight," and over 30 is "obese." The official definition uses meters and kilograms, but this implementation uses inches and pounds, so they have to be converted: one kilogram is 2.2 pounds, one inch is 2.54 centimeters.

bmi: compute body mass index
awk 'BEGIN { print "enter pounds inches" }
 { printf("%.1f\n", (\$1/2.2) / (\$2 * 2.54/100) ^ 2) } '
If we put this into an executable file bmi, we can run it as a program. For one author,

```
$ bmi
enter pounds inches
190 74
24.4
```

so he's (barely) "normal."

It's easy to run sensitivity tests as well, perhaps to assess the effects of dieting or measurement error:

```
$ bmi
enter pounds inches
195 74
25.1
200 75
25.1
```

Units Conversion

Are the conversion factors in the BMI example correct? If you don't remember them, the Unix program units provides conversion factors for hundreds of units:

The cf program below does a handful of common metric conversions with a more convenient user interface: provide a number as a command-line argument and cf converts it into temperature, length and weight. Rather than asking the user to specify a particular conversion, cf simply prints all conversions in both directions. The cf program began life as a Celsius to Fahrenheit temperature converter, which explains its name. For example, the outside temperature right now is 7°C; what's that in Fahrenheit?

\$ *cf* 7 7 C = 44.6 F; 7 F = -13.9 C

More conversions were added over time, such as lengths and weights. How much is 74 inches in centimeters, or 74 kg in pounds?

```
$ cf 74
74 C = 165.2 F; 74 F = 23.3 C
74 cm = 29.1 in; 74 in = 188.0 cm
74 kg = 162.8 lb; 74 lb = 33.6 kg
```

The program produces all the conversions; the user selects the desired one.

Here's the program. It uses the built-in array ARGV to access the single command-line argument. All the code is in the BEGIN block; it does not try to read any files.

```
# cf: units conversion for temperature, length, weight
awk 'BEGIN {
    t = ARGV[1] # first command-line argument
    printf("%s C = %.1f F; %s F = %.1f C\n",
        t, t*9/5 + 32, t, (t-32)*5/9)
    printf("%s cm = %.1f in; %s in = %.1f cm\n",
        t, t/2.54, t, t*2.54)
    printf("%s kg = %.1f lb; %s lb = %.1f kg\n",
        t, 2.2*t, t, t/2.2)
}' S*
```

The s^* in the script is the shell notation for the arguments that the program was called with; the shell expands it into a list of strings that are passed to the program. Most often these are filenames, but for cf the first one is the input number to be converted and any others are ignored.

Awk stores the arguments that the program was called with in the array ARGV, where ARGV[1] is the first argument, ARGV[2] the second, and so on to ARGV[ARGC-1]. ARGC is the number of arguments and ARGV[0] is the name of the program, usually awk. There's more information about ARGV in later examples and in Section A.5.5 of the reference manual.

A Reminder About Shell Scripts

The programs bmi and cf are *shell scripts*: programs written in a scripting language and stored in an executable file, so they can be invoked exactly the same as if they had been written in a compiled language like C. To make a file executable on Unix systems, run chmod ("change mode") once:

```
$ chmod +x bmi cf
```

If the script files are placed in a directory that is in your shell search path (which is often \$HOME/bin), you can use them as if they were built-in commands, as we have here.

2.2 Selection

The basic Awk structure is a set of patterns that select interesting lines and perform some actions on them. Many such programs are one-off, composed at the keyboard and run only a few times. Some, however, are both useful and too complicated to re-type each time, and are thus worth putting into a script that can be used whenever necessary.

Some examples duplicate existing Unix tools, so they are discussed here more for instructional value than to replace something, though Awk's flexibility means that you can make versions tailored to your specific needs. (It also means that you can have a portable version that doesn't differ from one flavor of Unix to another, as sometimes happens.) For instance, the Unix command head, which by default prints the first 10 lines of its input, is already equivalent to a simple sed command, 10q. The following Awk one-liner provides the same functionality.

NR <= 10

It isn't efficient on large inputs, however, because it reads the entire input but does nothing with it after the tenth line. An improved version would print each line but exit after the tenth:

```
{ print }
NR > 10 { exit }
```

The original version takes time proportional to the length of the input; the improved version takes a constant short time.

What if you want a program that will print the first three and the last three lines of its input, for example to see the most common and least common values in a numerically sorted list? One easy option is to store the entire input and print only the desired lines, like this:

```
awk '{ line[NR] = $0 }
END { for (i = 1; i <= 3; i++) print line[i]
print "..."
for (i = NR-2; i <= NR; i++) print line[i]
} ' $*</pre>
```

This isn't correct for inputs shorter than 7 lines, but as a personal tool, that may not matter as long as you remain aware of the limitation.

Since it stores the entire input to print only a small part, it might be slow for large files. Another approach is to print the first three lines as they arrive and then retain only the most recent three lines, printing them at the end:

```
awk 'NR <= 3 { print; next }
    { line[1] = line[2]; line[2] = line[3]; line[3] = $0 }
END { print "..."
    for (i = 1; i <= 3; i++) print line[i] } ' $*</pre>
```

The next statement stops processing the current record and starts processing the next one, starting with the first statement of the Awk program.

Somewhat surprisingly, this version is about one third slower than the first, perhaps because it's copying a lot of lines. A third option would be to treat the input as a circular buffer: store only three lines and cycle an index from 1 to 2 to 3 to 1 for the last three lines so there's no extra copying:

```
awk 'NR <= 3 { print; next }
    { line[NR%3] = $0 }
END { print "..."
    i = (NR+1) % 3
    for (j = 0; j < 3; j++) {
        print line[i]
        i = (i+1) % 3
    }
} ' $*</pre>
```

Empirically, this is only slightly faster than the first version, and the tiny improvement comes at the price of some tricky indexing in the END block. The added complexity can't be worth it for a program that takes only a few seconds to read and process a million lines of input.

There's always a tradeoff between processing input on the fly and collecting it in an array to be processed in the END block. Fortunately, modern processors are so fast and memories so capacious that it's usually fine to start with the simplest code possible rather than trying to save time or space. Certainly this is the case when you're evolving a program: simple first, faster but more complicated later but only if necessary.

A variation discards the first n lines and prints the rest; this is useful if there's a header line that identifies the fields of the remaining lines, and you want to get rid of it before subsequent processing:

awk 'NR > 1'

As it is for head, so it is for tail, which prints the last n lines of its input. This is a good example where you might want a version of your own, because not all versions offer the particularly useful option of printing the last lines in reverse order. Here's a simple version, called tail-r, that reads in the entire file, like head above, then prints the last three lines in reverse order:

```
awk '{ line[NR] = $0 }
END { for (i = NR; i > NR-3; i--) print line[i] } ' $*
```

If the test i > NR-3 is replaced by i > 0, the program can be used to reverse an entire file.

Exercise 2-1. Fix the program that prints the first few and last few lines of its input so that it works correctly even on short inputs. \Box

Exercise 2-2. Make a version of the line-reversal program above where the number of lines to print is a parameter. \Box

2.3 Transformation

Transforming input into output is what computers do, but there's a specific kind of transformation that Awk is meant for: text data enters, some modest change is made to all or some selected lines, and then it leaves.

Carriage Returns

One example comes from the unfortunately (and unnecessarily) different way that lines are terminated on Windows versus macOS and Unix. On Windows, each line of a text file ends with a carriage return character \r and a newline character \n , where on macOS and Unix, each line ends with a newline only. For a variety of good reasons (including the authors' cultural heritage and experience with Unix since its earliest days), Awk uses the new-line-only model, though if input is in Windows format, it will be processed correctly.

We can use one of Awk's text substitution functions, sub, to eliminate the r on each line. The function sub (*re*, *repl*, *str*) replaces the first match of the regular expression *re* in *str* by the replacement text *repl*. If there is no *str* argument, the replacement is done in \$0. Thus

{ sub(/\r\$/, ""); print }

removes any carriage return at the end of a line before printing.

The function gsub is similar but replaces all occurrences of text that match the regular expression; g implies "global." Both sub and gsub return the number of replacements they made so you can tell whether anything changed.

Going in the other direction, it's easy to insert a carriage return before each newline if there isn't one there already:

```
{ if (!/\r$/) sub(/$/, "\r"); print }
```

The test succeeds if the regular expression does not match, that is, if there is no carriage return at the end of the line. Regular expressions are covered in great detail in Section A.1.4 of the reference manual.

Multiple Columns

Our next example is a program that prints its input in multiple columns, under the assumption that most lines are shortish, for example a list of filenames or the names of people. For example, we would like to convert a sequence of names like

Alice Archie Eva Liam Louis Mary Naomi Rafael Sierra Sydney

into something like this:

Alice Archie Eva Liam Louis Mary Naomi Rafael Sierra Sydney

This program presents a variety of design choices, of which we will explore two; others make fine exercises, especially if your needs differ.

One basic choice is whether to read the entire input to figure out the range of sizes, or to produce the output on the fly without knowing what is to come. Another choice is whether to print the output in row order (which is what we'll do) or in column order; if the latter, then necessarily all the input has to be read before any output can be produced.

Let's do a streaming version first. The program assumes that input lines are no more than 10 characters wide, so with a couple of spaces between columns, there's room for 5 columns in a 60-character line. We can truncate too-long lines, with or without an indicator, or we can insert ellipses in the middle, or we can print them across multiple columns. Such choices could be parameterized, though that's definitely overkill for what is meant to be a simple example.

Here's a streaming version that silently truncates input lines; it's probably easiest:

```
# mc: streaming version of multi-column printing
{ out = sprintf("%s%-10.10s ", out, $0)
    if (n++ > 5) {
        print substr(out, 1, length(out)-2)
        out = ""
        n = 0
    }
}
END {
    if (n > 0)
        print substr(out, 1, length(out)-2)
}
```

The function sprintf is a version of printf that returns a formatted string rather than printing to an output stream. The sprintf conversion \$-10.10s is used to truncate the string and left justify it in a field of width 10.

The second line

if (n++ > 5)

illustrates a subtle but important point about the ++ operator. When it is written after the variable ("postfix"), the value of the expression is the value of the variable before it is incremented; the variable is incremented after. The prefix form ++n increments first and then returns the value.

Here's an alternate multi-column printer that collects the entire input and computes the widest field. It uses that width to create a suitable printf string, then formats the output into wide-enough columns. Note the use of %% to create a literal % in a format.

```
# mc: multi-column printer
\{ lines[NR] = \$0 \}
  if (length(\$0) > max)
    max = length(\$0)
}
END {
  fmt = sprintf("%%-%d.%ds", max, max) # make a format string
  ncol = int(60 / max + 0.5) # int(x) returns integer value of x
  for (i = 1; i <= NR; i += ncol) {
    out = ""
    for (j = i; j < i+ncol && j <= NR; j++)</pre>
      out = out sprintf(fmt, lines[j]) "
    sub(/ +$/, "", out) # remove trailing spaces
    print out
  }
}
```

The three lines

```
for (j = i; j < i+ncol && j <= NR; j++)
   out = out sprintf(fmt, lines[j]) "
   sub(/ +$/, "", out) # remove trailing spaces</pre>
```

append lines and spaces to the end of out, which is building up the output line. When the loop is finished, sub removes any trailing spaces; the regular expression matches one or more spaces at the end of the line. It would certainly be possible to avoid putting them there in the first place, but it's often simpler to add them, then remove them at the end, as we have done with both of the multi-column examples.

Exercise 2-3. Implement some of the parameterizations suggested above.

2.4 Summarization

Awk is useful for getting a quick summary of files that contain tabular data: maximum and minimum values, sums of columns, and the like. These are useful for data validation — what is in each field, whether there are empty values, and so on. This section contains several examples, and there are more in the next chapter, which discusses exploratory data analysis.

The addup script adds up the values in each column of its input and reports the sums at the end. It's a simple exercise in array subscripts:

```
# addup: add up values in each field separately
{ for (i = 1; i <= NF; i++)
    field[i] += $i
    if (NF > maxnf)
    maxnf = NF
}
END {
    for (i=1; i <= maxnf; i++)
        printf("%6g\t", field[i])
    printf("\n")
}</pre>
```

As mentioned earlier, the += operator in the second line increments the variable on the left of the equals sign by the value of the expression on the right. It's called an *assignment operator* and that statement is a shorthand for field[i] = field[i] + \$i. All of the arithmetic operators allow this shorthand form.

What if some value or even an entire column is not numeric? No problem. Awk uses the numeric prefix of a string as its numeric value, so if a value doesn't have a numeric-appearing prefix, its value is zero. For example, the numeric value of a string like "50% off" is 50.

Our personal repertoire of scripts includes variations on the addup theme, for computing minimum and maximum values in each field, computing simple statistical summaries like mean and variance, counting non-empty values, displaying the most common and least common entries, and so on, all of which are useful for getting quick insights into the properties of data, or for looking for anomalies and potential errors.

Some spreadsheet tools provide similar functionality, as in Google Sheets, and so does the Pandas library for Python. The advantage of using Awk is that you can tailor the computation to your specific need; naturally the corresponding disadvantage is that you have to write a bit of code yourself.

2.5 Personal Databases

Another Awk application area is the maintenance of personal databases. If you're a fitness person, you might already be keeping track of how far you've walked or run every day, what your weight is, and other numbers of interest. There are plenty of apps for keeping track, with nice interfaces, pretty graphics, and great charts. The potential drawback for some of these is the invasion of privacy, and of course there's always the question of whether they do exactly what you want.

One alternative is to keep the data in a plain text file and process it with Awk and other tools. Here's a simple example. Suppose you want to keep track of how many steps you walk, with the goal of getting to at least 10,000 every day. Create a file steps, in which each line has two fields, a date and a number, like this:

6/24/23	9342
6/25/23	4493
6/26/23	4924
6/27/23	16611
6/28/23	8762
6/29/23	15370
6/30/23	17897
7/1/23	6087
7/2/23	7595
7/3/23	14347
7/4/23	15762
7/5/23	20021

These are actual numbers for one of the authors, who was on vacation in a great place for walking, but only when the weather was good.

You can decide whether it's easier to enter your new data at the beginning of the file or the end; we've chosen to use the end.

Either way, you can write scripts to compute the average number of steps per time period. Here's a slightly ornate version that computes sliding-window averages for 7 days, 30 days, 90 days, one year, and lifetime:

```
awk '
{ s += $2; x[NR] = $2 }
END {
for (i = NR-6; i <= NR; i++) w += x[i]
for (i = NR-30; i <= NR; i++) m += x[i]
for (i = NR-90; i <= NR; i++) q += x[i]
for (i = NR-365; i <= NR; i++) yr += x[i]
printf(" 7: %.0f 30: %.0f 90: %.0f 1yr: %.0f %.1fyr: %.0f\n",
    w/7, m/30, q/90, yr/365, NR/365, s/NR)
} ' $*</pre>
```

It produces output like this:

```
7: 9679 30: 11050 90: 11140 1yr: 10823 13.7yr: 10989
```

Text files work well for medical data (weight, blood sugar, blood pressure), personal finance (stock prices, portfolio value), and many other areas. There are real advantages to a simple flat file processed by Awk and similar tools: the data is yours, not someone else's; it's easily updated with your favorite text editor; and it can be processed in ways that you might not have thought of originally.

For example, a simple extension to the step counter produces a histogram that shows how often you walked different distances, giving you an idea of how uniform or irregular your exercise habits might be:

```
awk '
{ s += $2; x[NR] = $2; dist[int($2/2000)]++ }
END {
  for (i = NR-6; i \le NR; i++) w += x[i]
  for (i = NR-30; i <= NR; i++) m += x[i]
  for (i = NR-90; i <= NR; i++) q += x[i]
  for (i = NR-365; i <= NR; i++) yr += x[i]
  printf(" 7: %.0f 30: %.0f 90: %.0f 1yr: %.0f %.1fyr: %.0f\n",
    w/7, m/30, q/90, yr/365, NR/365, s/NR)
  scale = 0.05
  for (i = 1; i <= 10; i++) {
    printf("%5d: ", i*2000)
    for (j = 0; j < scale * dist[i]; j++)</pre>
      printf("*")
    printf("\n")
  }
} ' $*
```

This program prints rows of asterisks; the number of asterisks is proportional to the number of days when you walked that many steps.

2000:	* * * *
4000:	**********
6000 :	*********
8000:	* * * * * * * * * * * * * * * * * * * *
10000:	* * * * * * * * * * * * * * * * * * * *
12000:	* * * * * * * * * * * * * * * * * * * *
14000:	* * * * * * * * * * * * * * * * * * * *
16000:	* * * * * * * * * * * * * * * * * * * *
18000:	*****
20000:	*

This doesn't work right if the distance is more than 20,000 steps, and it needs enough data before the longer time periods become meaningful. At some point the output lines also become too long, so the program has a scale factor to keep them within bounds.

Realistically, one isn't going to use this kind of plot often, when there are so many really good plotting packages available. In Section 7.2 we'll show a program to generate a Python program that does nice plots. And it's standard practice to generate a file with fields separated by commas and use Excel, Google Sheets, or the like to create high-quality charts, though the process may require manual steps. But Awk is a good way to manage and massage the data before it gets passed on to other tools.

Stock Prices

There's another kind of personal data of great interest to many people: investments. Where is my money and how well is it doing? This subsection shows one *ad hoc* example, a script that scrapes the prices of a list of stock ticker symbols from a web page.

Web scraping is a common application. Some web site has the information you want, but in the wrong form, and you want to extract the information once or periodically. We'll illustrate by scraping stock prices, but the same approach will work for a wide variety of other kinds of data. Web pages are formatted for human viewers, so the task for a program is to remove the formatting while preserving the relevant information. This operation might be easier with a proper HTML parser like the excellent BeautifulSoup library for Python, but Awk has the advantage of being already installed and easy to get started with.

The site we use is bigcharts.marketwatch.com, but of course this may not work by the time you read this. The specific web page and query is

```
bigcharts.marketwatch.com/quotes/multi.asp?view=q&msymb=tickers
```

where *tickers* is one or more ticker symbols separated by plus signs, like this:

```
$ quote aapl+amzn+fb+goog
AAPL 134.76
AMZN 98.12
FB 42.75
GOOG 92.80
$
```

We use the invaluable Unix program curl to retrieve the web page, then use Awk to discard all the HTML. Finding the useful bits is empirical: study sample output and use regular expression substitutions to eliminate the dross. Fortunately this site is clean and systematic so it's easy. Note how the long quoted argument is split into two lines with a backslash.

```
# guote - retrieve stock guotes for a list of tickers
curl "https://bigcharts.marketwatch.com/quotes/\
multi.asp?view=q&msymb=$1" 2>/dev/null |
awk '
 /
    sub(/.*/, "")
    sub(/<.*/, "")
    symb = $0
    next
 }
 /
    sub(/.*/, "")
    sub(/<.*/, "")</pre>
   price = $0
   gsub(/,/, "", price)
   printf("%6s %s\n", symb, price)
 }
.
```

The command-line argument, a list of tickers, is passed to curl in the shell parameter \$1. The construction 2>/dev/null is a shell idiom that discards the progress report output from curl; it's optional here.

Exercise 2-4. Write your own version of a stock-tracking program. Make it write output in CSV format so the data can be loaded into Excel or the like for plotting. \Box

2.6 A Personal Library

Awk provides a modest library of built-in functions like length, sub, substr, printf, and a dozen or two more; they are listed in Section A.2.1 of the reference manual. It's possible to create more functions of your own, to be included in an Awk program when

you need them. One good example would be a function that uses sub or gsub, but returns the modified string rather than a count. In this section we'll look at a handful of others that we have found useful over the years.

The function rest (n) returns the rest of the input fields, starting from the *n*-th:

```
# rest(n): returns fields n..NF as a space-separated string
function rest(n, s) {
  s = ""
  while (n <= NF)
    s = s $n++ " "
  return substr(s, 1, length(s)-1) # remove trailing space
}
# test it:
{ for (i = 0; i <= NF+1; i++)
    printf("%3d [%s]\n", i, rest(i))
}
```

The function rest has a local variable, s. In Awk, there are no declarations, so (a bad design, sadly) any parameters that are not provided by the caller of the function are assumed to be local variables in the function. For example, rest is called with only one argument, n, so the second parameter, s, is a local variable within the function.

We conventionally write function declarations with several extra spaces in front of the local variable names to make it somewhat clearer what their role is. An alternative is to use distinctive names, for example leading or trailing underscores:

```
function rest(n, _s) {
   _s = ""
   while (n <= NF)
   _s = _s $n++ " "
   return substr(_s, 1, length(_s)-1)
}</pre>
```

though this is visually unappealing.

Yet another option is to add an unused parameter with a name like locals or _ at the beginning of the list of locals. All of these are imperfect workarounds for a poor language design.

Variations on the rest theme could include subfields(m, n) to return a contiguous sequence of fields *m* through *n*, or a join function that returns all the values in an array as a sequence separated by spaces, or a different one that converts an array into a JSON object like

```
{"name": "value", ...}
```

If you use the standard Awk, you have to manually copy such functions into your program, which is the moral equivalent of cutting and pasting: easy but with some risks. There is also an include program in Section A.5.4 of the reference manual. Or you could use multiple -f arguments to include multiple Awk source files.

Date Formatter

The dates in the example at the beginning of the previous section are in the form mm/dd/yy, which is conventional in the USA but different elsewhere, and also irregular and hard to sort or do arithmetic on. It's easy to write a datefix function that converts this date

format into the ISO standard format yyyy-mm-dd so that data can be directly sorted into date order.

```
# datefix: convert mm/dd/yy into yyyy-mm-dd (for 1940 to 2039)
awk '
function datefix(s, y, date) {
   split(s, date, "/")
   y = date[3]<40 ? 2000+date[3] : 1900+date[3] # arbitrary year
   return sprintf("%4d-%02d-%02d", y, date[1], date[2])
}
{ print(datefix($0)) }
' $*
$ datefix
12/25/23
2023-12-25</pre>
```

The built-in function split(s, arr, sep) uses the separator *sep* to split the string *s* into an array *arr*. The elements are numbered starting from 1, and split returns the number of elements. The separator is a regular expression, and can be written as a string like "*sep*" or within slashes like */sep/*. If there is no *sep* argument, then if -csv is set, fields are split as CSV; otherwise, the value of the field-separator variable FS is used. (See Section A.5.2.)

There is one special case: if *sep* is the empty string "" or empty regular expression //, the string is split into its individual characters, one array element per character.

The code converts a 2-digit year into four digits with an arbitrary rule: if it's less than 40, assume the year is 20xx, otherwise 19xx.

The ?: operator has the syntax $expr_1$? $expr_2$: $expr_3$, as in C. It evaluates $expr_1$. If it is true, the result is $expr_2$, otherwise $expr_3$; only one of these is evaluated. In effect ?: is a compact if-else that can be used within an expression. It's convenient, but easily abused to make inscrutable code.

Finally, note the conversions in sprintf: %02d prints an integer in a 2-digit field, padding it with a leading zero if necessary.

Suppose we want to get the current date and time from the local operating system. We can use the Unix date command, then reformat its contents. The easiest way is to run date and pipe its output into the Awk getline function, which reads inputs from files or pipes:

```
"date" | getline date  # get current date and time
split(date, d, / /)  # or equivalently, " "
date = d[2] " " d[3] ", " d[6]
```

A little processing converts the date from this format:

Wed Jul 12 07:16:19 EDT 2023

into this:

Jul 12, 2023

or any other desired format, perhaps using datefix.

The getline command and input pipes are covered in more detail in Section A.5.4.

Suppose you want to convert month names into numbers, so that Jan becomes 1, Feb becomes 2, and so on. This can be done with a sequence of assignments like m["Jan"]=1,

m["Feb"] = 2, and so on, but it's tedious if there are more than a handful of assignments. A useful alternative is a function that splits a string into an indexed array, like this:

```
# isplit - make an indexed array from str
function isplit(str, arr, n, i, temp) {
    n = split(str, temp)
    for (i = 1; i <= n; i++)
        arr[temp[i]] = i
    return n
}
```

The function isplit is like split, except that it creates an array whose *subscripts* are the words within the string, and whose values are the indices of the words in the string. Thus after

isplit ("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec", m)

the value of m["Jan"] is 1 and m["Dec"] is 12.

The split function can have a third argument that is a regular expression, and that could be passed in to isplit (as a literal string) for other applications.

Exercise 2-5. Write a script tomorrow that prints tomorrow's date in a suitable format.

Exercise 2-6. Write versions of sub and gsub that return the modified string, analogous to the re.sub function of Python. \Box

2.7 Summary

In this chapter, we've shown a bunch of scripts that we have personally found useful. The odds are good that most of them won't be directly what you want, but we hope that they have given you some ideas for your own programs, and have also illustrated a variety of techniques that will make your programming easier.

Most of the examples are based on some combination of arithmetic expressions for computing relevant values, arrays for storing information, and functions for encapsulating computation. These mechanisms are of fundamental importance in programming. They are particularly easy to use in Awk, which is designed around them, but the same approaches are invaluable in other languages as well, and it's well worth your while to become proficient in them.

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.

Туре	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:

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
}</pre>
```

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. \Box

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. \Box

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:

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:

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</pre>
```

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 nis 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
е	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 \mathbb{R} (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

4

Data Processing

Awk was originally intended for everyday data processing, such as information retrieval, data validation, and data transformation and summarization like those in the three previous chapters. In this chapter, we will consider similar but more complex tasks. Most of the examples deal with the usual line-at-a-time processing, but the final section describes how to handle data where an input record may occupy several lines.

Awk programs are often developed incrementally: a few lines are written and tested, then a few more added, and so on. The longer programs in this book were developed in this way.

It's also possible to write Awk programs in the traditional way, sketching the outline of the program, consulting the language manual, and so forth. But modifying an existing program to get the desired effect is frequently easier. The programs in this book thus serve as useful models for programming by example.

4.1 Data Transformation and Reduction

One of the most common uses of Awk is to transform data from one form to another, usually from the form produced by one program to a different form required by some other program. Another use is selection of relevant data from a larger dataset, often with reformatting and the preparation of summary information. This section contains a variety of examples of these topics.

Summing Columns

We have already seen several variants of the two-line Awk program that adds up all the numbers in a single field, and in Chapter 2, we saw an addup program that added up the numbers in each field separately. That program didn't check anything about the fields. The following program performs a somewhat more complicated but still representative data-reduction task. Every input line has several fields, each containing numbers, and the task is to compute the sum of each column of numbers, regardless of how many columns there are.

It's convenient that the program doesn't need to be told how many fields a row has, but it doesn't check that the entries are all numbers, nor that each row has the same number of entries. This version of addup does the same job, but also checks that each row has the same

number of entries as the first:

The printf in the END action shows how a conditional expression can be used to put tabs between the column sums and a newline after the last sum.

Now suppose that some of the fields are nonnumeric, so they shouldn't be included in the sums. One strategy is to add an array numcol to keep track of which fields are numeric, and a function isint to check if an entry is a number. We made it a function so the test occurs in only one place, in anticipation of future changes. If the program can trust its input, it need only look at the first line to tell if a field will be numeric.

```
# addup3 - print sums of numeric columns
#
      input: rows of integers and strings
        assumes every line has same layout
#
#
      output: sums of numeric columns
NR==1 { nfld = NF
        for (i = 1; i \le NF; i++)
            numcol[i] = isint($i)
      { for (i = 1; i <= NF; i++)
            if (numcol[i])
                sum[i] += $i
END
      { for (i = 1; i <= nfld; i++) {
            if (numcol[i])
                printf("%q", sum[i])
            else
                printf("--")
            printf(i < nfld ? "\t" : "\n")</pre>
        }
      }
```

function isint(n) { return n ~ $/^{[+-]?[0-9]+$/ }}$

The function isint defines an integer as one or more digits, perhaps preceded by a sign. A more general definition for floating-point numbers can be found in the discussion of regular expressions in Section A.1.4 of the reference manual.

Exercise 4-1. Modify addup3 to ignore blank lines.

Exercise 4-2. Add the more general regular expression for a number. How does it affect the running time? \Box

Exercise 4-3. What is the effect of removing the test of numcol in the second for statement?

Computing Percentages and Quantiles

Suppose that we want not the sum of a column of numbers but what percentage each is of the total. This requires two passes over the data. If there's only one column of numbers and not too much data, the easiest way is to store the numbers in an array on the first pass, then compute the percentages on the second pass as the values are being printed:

```
# percent - compute percentage each input number represents
# input: a column of nonnegative numbers
# output: each number and its percentage of the total
    { x[NR] = $1; sum += $1 }
END { if (sum != 0)
        for (i = 1; i <= NR; i++)
            printf("%10.2f %5.1f\n", x[i], 100*x[i]/sum)
}</pre>
```

This same approach, though with a more complicated transformation, could be used, for example, in adjusting student grades to fit some curve. Once the grades have been computed as numbers between 0 and 100, it might be interesting to see a histogram:

Note how the postfix decrement operator -- is used to control the while loop in rep.

We can test histogram with randomly generated grades. The first program in the pipeline below generates 200 random numbers between 0 and 100, and pipes them into the histogram maker. (The function rand returns a value greater than or equal to 0 and less than 1).

It produces this output:

0	_	9.	17	* * * * * * * * * * * * * * * *
10		10	1 / 2 2	
10	_	19:	23	
20	-	29:	20	* * * * * * * * * * * * * * * * * * * *
30	-	39:	15	* * * * * * * * * * * * * *
40	-	49:	15	* * * * * * * * * * * * * *
50	-	59:	21	* * * * * * * * * * * * * * * * * * * *
60	-	69:	19	* * * * * * * * * * * * * * * * * * *
70	-	79:	19	* * * * * * * * * * * * * * * * * * *
80	-	89:	22	* * * * * * * * * * * * * * * * * * * *
90	-	99:	25	* * * * * * * * * * * * * * * * * * * *
100:	:		4	* * * *

Exercise 4-4. Make a version of the histogram code that divides the input into a specified number of buckets, adjusting the ranges according to the data seen. \Box

Numbers with Commas

Suppose we have a list of numbers that contain commas and decimal points, like 12, 345.67. Since Awk thinks that the first comma terminates a number, these numbers cannot be summed directly. The commas must first be erased:

```
# sumcomma - add up numbers containing commas
{ gsub(/,/, ""); sum += $1 }
END { print sum }
```

The first argument of gsub is a regular expression, in this case one that matches a comma. The effect of gsub(/,/, "") is to replace every comma in \$0 with the null string, that is, to delete the commas.

This program doesn't check that the commas are in the right places, nor does it print commas in its answer. Putting commas into numbers requires only a little effort, as the next program shows. It formats numbers with commas and two digits after the decimal point. The structure of this program is a useful one to emulate: it contains a function that only does the new thing, with the rest of the program just reading and printing. After it's been tested and is working, the new function can be included in the final program.

```
# addcomma - put commas in numbers
   input: a number per line
#
#
   output: the input number followed by
#
       the number with commas and two decimal places
    { printf("%-12s %20s\n", $0, addcomma($0)) }
function addcomma(x,
                      num) {
   if (x < 0)
       return "-" addcomma(-x)
   num = sprintf("%.2f", x)  # num is dddddd.dd
   while (num \sim /[0-9][0-9][0-9]])
        sub(/[0-9][0-9][0-9][,.]/, ",&", num)
   return num
}
```

The basic idea is to insert commas from the decimal point to the left in a loop; each iteration puts a comma in front of the leftmost three digits that are followed by a comma or decimal point, provided there will be at least one additional digit in front of the comma. The algorithm uses recursion to handle negative numbers: if the input is negative, the function addcomma calls itself with the positive value, tacks on a leading minus sign, and returns the result. Note the & in the replacement text for sub to add a comma before each triplet of numbers.

Here are the results for some test data:

0	0.00
-1	-1.00
.1	0.10
-12.34	-12.34
-12.345	-12.35
12345	12,345.00
-1234567.89	-1,234,567.89
-123.	-123.00
-123456	-123,456.00

Exercise 4-5. Modify sumcomma, the program that adds numbers with commas, to check that the commas in the numbers are properly positioned. \Box

Fixed-Field Input

Information appearing in fixed-width fields may require preprocessing before it can be used directly. Some programs print information in fixed columns, rather than with field separators; if the fields are too wide, the columns abut.

Fixed-field data is best handled with substr, which can be used to pick apart any combination of columns. For example, the output of the Unix 1s command is formatted so everything lines up:

```
total 3024
drwxr-xr-x 9 bwk staff 288 Mar 7 2019 Album Artwork
drwxr-xr-x 4 bwk staff 128 Mar 7 2019 Previous iTunes Libraries
-rw-r--r--@ 1 bwk staff 73728 Jul 3 19:34 iTunes Library Extras.itdb
-rw-r--r--@ 1 bwk staff 32768 Jul 16 2016 iTunes Library Genius.itdb
-rw-r--r--@ 1 bwk staff 1377841 Jul 3 19:34 iTunes Library.itl
drwxr-xr-x 6 bwk staff 192 May 15 2020 iTunes Media
-rw-r--r--@ 1 bwk staff 8 Jul 3 19:34 sentinel
```

The filenames are at the end but because they may contain spaces, they have to be extracted either with a function like rest (9), which we described in Section 2.6, or with substr, as in this example:

```
{ print substr($0, index($0, $9)) }
total 3024
Album Artwork
Previous iTunes Libraries
iTunes Library Extras.itdb
iTunes Library Genius.itdb
iTunes Library.itl
iTunes Media
sentinel
```

Notice the use of index to compute what column the filename begins in.

Exercise 4-6. This code doesn't work if the filename in \$9 also appears as a substring earlier on the line. Fix it. \Box

Program Cross-Reference Checking

Awk is often used to extract information from the output of other programs. Sometimes that output is merely a set of homogeneous lines, in which case field-splitting or substr operations are sufficient. Sometimes, however, the upstream program thinks its output is intended for people. In that case, the task of the Awk program is to undo careful formatting, so as to extract the information from the irrelevant. The next example is a simple instance.

Large programs are built from many files. It is convenient (and sometimes vital) to know which file defines which function, and where the function is used. To that end, the Unix program nm prints a neatly formatted list of the names, definitions and addresses, and uses of the names in a set of object files. A typical fragment of its output looks like this:

```
lex.o:
0000000000000000 T startreg
U strcmp
0000000000003d0 T string
U strlen
U strtod
lib.o:
0000000000002f0 T eprint
00000000000015f0 T errcheck
000000000000680 T error
U exit
U fclose
```

Lines with one field (e.g., lex.o) are filenames, lines with two fields (e.g., U and fclose) are uses of names, and lines with three fields are definitions of names. T indicates that a definition is a text symbol (function) and U indicates that the name is undefined.

Using this raw output to determine what file defines or uses a particular symbol can be a nuisance, because the filename is not attached to each symbol. For a C program the list can be long — it's 750 lines for the nine files of source that make up Awk itself. A three-line Awk program, however, can add the name (without the colon) to each item:

```
# nm.format - add filename to each nm output line
NF == 1 { sub(/:/,""); file = $1 }
NF == 2 { print file, $1, $2 }
NF == 3 { print file, $2, $3 }
```

The output from nm.format on the data shown above is

lex.o T startreg lex.o U strcmp lex.o T string lex.o U strlen lex.o U strtod lib.o T eprint lib.o T error lib.o U exit lib.o U fclose

Now it's easy for other programs to search this output or process it further.

This technique does not tell us where or how many times a name appears in a file, but these things can be found by a text editor or another Awk program. Nor does it depend on which language the programs are written in, so it is more flexible than the usual run of crossreferencing tools, and shorter and simpler too.

4.2 Data Validation

Another common use for Awk programs is data validation: making sure that data is legal or at least plausible. We saw some specific examples in Chapter 3 when we were looking at data from the Titanic. This section describes several small general-purpose programs that check input for validity. For example, consider the column-summing programs in the previous section. Are there any numeric fields where there should be nonnumeric ones, or vice versa? Such a program is close to one we saw before, with the summing removed:

```
# colcheck - check consistency of columns
    input: rows of numbers and strings
#
#
    output: lines whose format differs from first line
NR == 1
        {
    nfld = NF
    for (i = 1; i \le NF; i++)
       type[i] = isint($i)
}
   if (NF != nfld)
{
       printf("line %d has %d fields instead of %d\n",
         NR, NF, nfld)
    for (i = 1; i <= NF; i++)
       if (isint($i) != type[i])
          printf("field %d in line %d differs from line 1\n",
             i, NR)
}
function isint(n) { return n ~ /^{[+-]?[0-9]+$/ }}
```

This certainly doesn't check for all possible errors. The test for integers is again just a sequence of digits with an optional sign; see the discussion of regular expressions in Section A.1.4 of the reference manual for a more complete explanation.

Balanced Delimiters

In the machine-readable text of this book, each program is introduced by a line beginning with .P1 and is terminated by a line beginning with .P2. These lines are text-formatting commands that make the programs come out in their distinctive font when the text is typeset. Since programs cannot be nested, these text-formatting commands must form an alternating sequence

.P1 .P2 .P1 .P2P1 .P2

If one or the other of these delimiters is omitted, the output will be badly mangled by our text formatter. To make sure that the programs will be typeset properly, we wrote this tiny delimiter checker, which is typical of a large class of such programs:
If the delimiters are in the right order, the variable p silently goes through the sequence of values 0 1 0 1 0 ... 1 0. Otherwise, the appropriate error messages are printed. We use a larger version to check the manuscript of the book for similar errors.

Exercise 4-7. What's a good way to extend this program to handle multiple sets of delimiter pairs or nested delimiters? \Box

Password-File Checking

The password file on a Unix system used to contain the names of and other information about authorized users. Each line of the password file had 7 fields, separated by colons:

```
root:qyxRi2uhuVjrg:0:2::/:
bwk:lL./v6iblzzNE:9:1:Brian Kernighan:/usr/bwk:
ava:otxsloTVoyvMQ:15:1:Al Aho:/usr/ava:
uucp:xutIBs2hKtcls:48:1:uucp daemon:/usr/lib/uucp:uucico
pjw:xNqy//GDc8FFg:170:2:Peter Weinberger:/usr/pjw:
...
```

The first field is the user's login name, which should be alphanumeric. The second is an encrypted version of the password; if this field is empty, anyone can log in pretending to be that user, while if there is a password, only people who know the password can log in. The third and fourth fields are supposed to be numeric. The sixth field, the user login directory, should begin with /. The following program prints all lines that fail to satisfy these criteria, along with the number of the erroneous line and an appropriate diagnostic message.

```
# checkpasswd - check password file for correct format
BEGIN { FS = ":" }
NF != 7 {
    printf("line %d, does not have 7 fields: %s\n", NR, $0) }
$1 ~ /[^A-Za-z0-9]/ {
    printf("line %d, nonalphanumeric user id: %s\n", NR, $0) }
$2 == "" {
    printf("line %d, no password: %s\n", NR, $0) }
$3 ~ /[^0-9]/ {
    printf("line %d, nonnumeric user id: %s\n", NR, $0) }
$4 ~ /[^0-9]/ {
    printf("line %d, nonnumeric group id: %s\n", NR, $0) }
$6 !~ /^\// {
    printf("line %d, invalid login directory: %s\n", NR, $0) }
```

This is a good example of a program that can be developed incrementally: each time someone thinks of a new condition that should be checked, it can be added, so the program steadily becomes more thorough.

Generating Data-Validation Programs

We constructed the password-file checking program by hand, but a more interesting approach is to convert a set of conditions and messages into a checking program automatically. Here is a small set of error conditions and messages, where each condition is a pattern from the program above. The error message is to be printed for each input line where the condition is true.

```
NF != 7 does not have 7 fields
$2 == "" no password
$1 ~ /[^A-Za-z0-9]/ nonalphanumeric user id
```

The following program converts these condition-message pairs into a checking program:

The output is a sequence of conditions and the actions to print the corresponding messages:

```
NF != 7 {
    printf("line %d, does not have 7 fields: %s\n",NR,$0) }
$2 == "" {
    printf("line %d, no password: %s\n",NR,$0) }
$1 ~ /[^A-Za-z0-9]/ {
    printf("line %d, nonalphanumeric user id: %s\n",NR,$0) }
```

When the resulting checking program is executed, each condition will be tested on each line, and if the condition is true, the line number, error message, and input line will be printed. Note that in checkgen, some of the special characters in the printf format string must be quoted to produce a valid generated program. For example, % is preserved by writing %% and \n is created by writing \\n.

This technique in which one Awk program creates another one is broadly applicable, and of course it's not restricted to Awk programs.

By the way, as a historical note, one of the inspirations for Awk was an error-checking tool created by Marc Rochkind at Bell Labs in the mid 1970s. Marc's program, written in C, took a sequence of regular expressions as input and created a C program that would scan its input and report any line that matched any of the patterns. It was a very neat idea, and we stole it unabashedly.

4.3 Bundle and Unbundle

Consider how to combine ("bundle") a set of text files into one file in such a way that they can be easily separated ("unbundled") into the original files. This section contains two tiny Awk programs that do this pair of operations. They can be used for bundling small files together to save disk space, or to package a collection of files for convenient emailing.

The bundle program is trivial, so short that you can just type it on a command line. All it does is prefix each line of the output with the name of the file, which comes from the built-in variable FILENAME.

```
# bundle - combine multiple files into one
{ print FILENAME, $0 }
```

The matching unbundle is only a little more elaborate:

```
# unbundle - unpack a bundle into separate files
$1 != prev { close(prev); prev = $1 }
{ print substr($0, index($0, " ") + 1) >$1 }
```

The first line of unbundle closes the previous file when a new one is encountered. If bundles don't contain many files (less than the limit on the number of simultaneously open files), closing the file isn't necessary.

By the way, the >\$1 in the last line of unbundle is not a relational operator, but causes the output to be written to a file whose name is stored in \$1.

There are other ways to write bundle and unbundle, but the versions here are the easiest, and for short files, reasonably space efficient. Another organization is to add a distinctive line with the filename before each file, so the filename appears only once.

Exercise 4-8. Note that bundle assumes that filenames do not contain spaces. Fix it to handle filenames with spaces. \Box

Exercise 4-9. Compare the speed and space requirements of these versions of bundle and unbundle with variations that use headers and perhaps trailers. Evaluate the tradeoff between performance and program complexity. \Box

4.4 Multiline Records

The examples so far have featured data where each record fits neatly on one line. Many other kinds of data, however, come in multiline chunks. Examples include address lists:

```
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021
212 555-4321
```

or bibliographic citations:

```
Donald E. Knuth
The Art of Computer Programming
Volume 4B: Combinatorial Algorithms, Part 2
Addison-Wesley, Reading, Mass.
2022
```

or personal databases:

```
Chateau Lafite Rothschild 1947
12 bottles at 12.95
```

It's easy to create and maintain such information if it's of modest size and regular structure; in effect, each record is the equivalent of an index card. Dealing with such data in Awk requires only a bit more work than single-line data does; we'll show several approaches.

Records Separated by Blank Lines

Imagine an address list, where each record contains on the first four lines a name, street address, city and state, and phone number; after these, there may be additional lines of other information. Records are separated by a single blank line:

```
Adam Smith

1234 Wall St., Apt. 5C

New York, NY 10021

212 555-4321

David W. Copperfield

221 Dickens Lane

Monterey, CA 93940

408 555-0041

work phone 408 555-6532

birthday February 2

Canadian Consulate

466 Lexington Avenue, 20th Floor

New York, NY 10017

1-844-880-6519
```

When records are separated by blank lines, they can be manipulated directly: if the record separator variable RS is set to the null or empty string (RS=""), each multiline group becomes a record. Thus

```
BEGIN { RS = "" }
/New York/
```

will print each record that contains New York, regardless of how many lines it has:

```
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021
212 555-4321
Canadian Consulate
466 Lexington Avenue, 20th Floor
New York, NY 10017
1-844-880-6519
```

When several records are printed in this way, there is no blank line between them, so the input format is not preserved. The easiest way to fix this is to set the output record separator ORS to a double newline $\n\$:

```
BEGIN { RS = ""; ORS = "\n\n" }
/New York/
```

Suppose we want to print the names and phone numbers of all Smith's, that is, the first and fourth lines of all records in which the first line ends with Smith. That would be easy if each line were a field. This can be arranged by setting FS to n:

```
BEGIN { RS = ""; FS = "\n" }
$1 ~ /Smith$/ { print $1, $4 } # name, phone
```

This produces

Adam Smith 212 555-4321

A newline character is always a field separator for multiline records, regardless of the value of FS. When RS is set to "", the field separator by default is any sequence of spaces and tabs, or newline. When FS is set to n, only a newline acts as a field separator.

RS can also be a regular expression; in that case, the regular expression is used to identify the breaks between records. So, for example, if records are separated by a line consisting only of dashes, as in

```
record 1
------ 2
----- 7
record 3
```

and so on, it would be easy to process them with

```
RS = " \setminus n - - + \setminus n"
```

so the separator could be three or more dashes.

Processing Multiline Records

If an existing program can process its input only by lines, we may still be able to use it for multiline records by writing two Awk programs. The first combines the multiline records into single-line records that can be processed by the existing program. Then, the second transforms the processed output back into the original multiline format.

To illustrate, let's sort our address list with the Unix sort command. The following pipeline sorts the address list by last name:

```
# pipeline to sort address list by last names
awk '
BEGIN { RS = ""; FS = "n" }
      { printf("%s!!#", x[split($1, x, " ")])
        for (i = 1; i \le NF; i++)
            printf("%s%s", $i, i < NF ? "!!#" : "\n")
      }
'$*|
sort |
awk '
BEGIN { FS = "!!#" }
     { for (i = 2; i <= NF; i++)
            printf("%s\n", $i)
        printf("\n")
      }
.
```

In the first program, the function split(\$1, x, ") splits the first line of each record into the array x and returns the number of elements created; thus, x[split(\$1, x, ")] is the entry for the last name. (This assumes that the last word on the first line really is the last name, so it won't work for names like John D. Rockefeller Jr.) For each multiline record, the first program creates a single line that consists of the last name, followed by the string !!#, followed by all the fields in the record separated by this string. Any other separator that does not occur in the data and that sorts earlier than the data could be used in place of the string !!#.

The program after the sort reconstructs the multiline records using this separator to identify the original fields.

Exercise 4-10. Modify the first Awk program to detect occurrences of the string !! # in the data.

Records with Headers and Trailers

Sometimes records are identified by a header and trailer, rather than by a record separator. Consider a simple example, again an address list, but this time each record begins with a header that indicates some characteristic, such as occupation, of the person whose name follows, and each record (except possibly the last) is terminated by a trailer consisting of a blank line:

```
accountant
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021
doctor - ophthalmologist
Dr. Will Seymour
798 Maple Blvd.
Berkeley Heights, NJ 07922
lawyer
David W. Copperfield
221 Dickens Lane
Monterey, CA 93940
doctor - pediatrician
Dr. Susan Mark
600 Mountain Avenue
Murray Hill, NJ 07974
```

A range pattern is the simplest way to print the records of all doctors:

/^doctor/, /^\$/

The range pattern matches records that begin with doctor and end with a blank line $(/^{\$}/ \text{matches a blank line})$.

To print the doctor records without headers, we can use

```
/^doctor/ { p = 1; next }
p == 1
/^$/ { p = 0 }
```

This program uses a variable p to control the printing of lines. When a line containing the desired header is found, p is set to one; a subsequent line containing a trailer resets p to zero, its default initial value. Since lines are printed only when p is set to one, only the body and trailer of each record are printed; other combinations are easily selected instead. This is similar to the earlier example that looked for balanced delimiters.

Name-Value Data

In some applications data may have more structure than can be captured by a sequence of unformatted lines. For instance, addresses might include a country name, or might not have a street address.

One way to deal with structured data is to add an identifying name or keyword to each field of each record. For example, here is how we might organize a checkbook in this format:

```
check 1021
     Champagne Unlimited
tο
amount 123.10
     1/1/2023
date
deposit
amount 500.00
date 1/1/2023
check 1022
date 1/2/2023
amount 45.10
    Getwell Drug Store
to
     medical
tax
check 1023
amount 125.00
   International Travel
tο
date 1/3/2023
amount 50.00
  Carnegie Hall
to
date
      1/3/2023
check 1024
tax
     charitable contribution
   American Express
to
check 1025
amount 75.75
     1/5/2023
date
```

We are still using multiline records separated by a single blank line, but within each record, every piece of data is self-identifying: each field consists of an item name, a tab, and the information. That means that different records can contain different fields, or similar fields in arbitrary order.

One approach for this kind of data is to treat it as single lines, with occasional blank lines as separators. Each line identifies the value it corresponds to, but they are not otherwise connected. So to accumulate the sums of deposits and checks, for example, we could simply scan the input for deposits and checks, like this:

```
# check1 - print total deposits and checks
/^check/
         { chk = 1; next }
/^deposit/ { dep = 1; next }
/^amount/ \{ amt = $2; next \}
/^$/
           { addup() }
END
           { addup()
             printf("deposits $%.2f, checks $%.2f\n",
                 deposits, checks)
           }
function addup() {
    if (chk)
        checks += amt
    else if (dep)
        deposits += amt
    chk = dep = amt = 0
}
```

which produces

deposits \$500.00, checks \$418.95

This is easy, and it works (on correct input) no matter what order the items of a record appear in. But it is delicate, requiring careful initialization, reinitialization, and end-of-file processing. Thus an appealing alternative is to read each record as a unit, then pick it apart as needed. The following program computes the same sums of deposits and checks, using a function to extract the value associated with an item of a given name:

```
# check2 - print total deposits and checks
                { RS = ""; FS = "n" }
BEGIN
/(^|\n)deposit/ { deposits += field("amount"); next }
/(^|\n)check/ { checks += field("amount"); next }
                { printf("deposits $%.2f, checks $%.2f\n",
END
                      deposits, checks)
                }
function field(name, i, f) {
    for (i = 1; i <= NF; i++) {
        split($i, f, "\t")
        if (f[1] == name)
            return f[2]
    }
    printf("error: no field %s in record\n%s\n", name, $0)
}
```

The function field(s) finds an item in the current record whose name is s; it returns the value associated with that name.

A third possibility is to split each field into an associative array and access that for the values. To illustrate, this program prints the check information in a more compact form:

1/1/2023 1021 \$123.10 Champagne Unlimited 1/2/2023 1022 \$45.10 Getwell Drug Store 1/3/2023 1023 \$125.00 International Travel 1/3/2023 1024 \$50.00 Carnegie Hall 1/5/2023 1025 \$75.75 American Express

The program is:

```
# check3 - print check information
```

```
BEGIN { RS = ""; FS = "\n" }
/(^|\n)check/ {
  for (i = 1; i <= NF; i++) {
    split($i, f, "\t")
    val[f[1]] = f[2]
  }
  printf("%8s %5d %8s %s\n",
    val["date"],
    val["check"],
    sprintf("$%.2f", val["amount"]),
    val["to"])
  delete val
}</pre>
```

In check3, note the use of sprintf to put a dollar sign in front of the amount; the resulting string is then right-justified by printf. It's often handy to use sprintf to create a string of some desired format to be used elsewhere in a program.

Exercise 4-11. Write a command lookup x y that will print from a known file all multiline records having the item name x with value y. \Box

4.5 Summary

In this chapter, we've presented programs for a variety of different data-processing applications: fetching information from address lists, computing simple statistics from numerical data, checking data and programs for validity, and so forth. There are several reasons why such diverse tasks are fairly easy to do in Awk. The pattern-action model is a good match to this kind of processing. The adjustable field and record separators accommodate data in a variety of shapes and formats; associative arrays are convenient for storing both numbers and strings; functions like split and substr are good at picking apart textual data; and printf is a flexible output formatter. In the following chapters, we'll see further applications of these facilities.

5

Reports and Databases

This chapter shows how Awk can be used to extract information and generate reports from data stored in files. The emphasis is on tabular data, but the techniques apply to more complex forms as well. The theme is the development of programs that can be used with one another. We will see a number of common data-processing problems that are hard to solve in one step, but easily handled by making several passes over the data.

The first part of the chapter deals with generating reports by scanning a single file. Although the format of the final report is of primary interest, there are complexities in the scanning too. The second part of the chapter describes one approach to collecting data from several interrelated files. We've chosen to do this in a fairly general way, by thinking of the group of files as a relational database. One of the advantages is that fields can have names instead of numbers.

5.1 Generating Reports

Awk can be used to select data from files and then to format the selected data into a report or a "dashboard" that provides summary information about an activity. We will often use a three-step process to generate reports: prepare, sort, format.

The preparation step involves selecting data and perhaps performing computations on it to obtain the desired information. The sort step is necessary if we want to display the data in some particular order. To perform this step we pass the output of the preparation program into the system sort command. The formatting step is done by a second Awk program that generates the desired report from the sorted data.

A Simple Report

Our dataset for this section is a file called countries:

Russia	16376	145	Europe
China	9388	1411	Asia
USA	9147	331	North America
Brazil	8358	212	South America
India	2973	1380	Asia
Mexico	1943	128	North America
Indonesia	1811	273	Asia
Ethiopia	1100	114	Africa
Nigeria	910	206	Africa
Pakistan	770	220	Asia
Japan	364	126	Asia
Bangladesh	130	164	Asia

The file contains data about the land area in square kilometers and the population in millions of the dozen most populous countries; the fields are separated by tabs. This dataset isn't the most interesting in the world, and you should not take the actual values as authoritative, but in a small way it's typical of a wide class, a mixture of text and numbers from which we want to select and compute.

Suppose we want a report giving the population, area, and population density of each country. We would like the countries to be grouped by continent, and the continents to be sorted alphabetically; within each continent the countries are to be listed in decreasing order of population density, like this:

CONTINENT	COUNTRY	POPULATION	AREA	POP. DEN.
Africa	Nigeria	206	910	226.4
Africa	Ethiopia	114	1100	103.6
Asia	Bangladesh	164	130	1261.5
Asia	India	1380	2973	464.2
Asia	Japan	126	364	346.2
Asia	Pakistan	220	770	285.7
Asia	Indonesia	273	1811	150.7
Asia	China	1411	9388	150.3
Europe	Russia	145	16376	8.9
North America	Mexico	128	1943	65.9
North America	USA	331	9147	36.2
South America	Brazil	212	8358	25.4

The first two steps in preparing this report are done by the program prep1, which, when applied to the file countries, determines the relevant information and sorts it:

The output is a sequence of lines containing five fields, separated by commas, that give the continent, country, population, area, and population density:

```
Africa, Nigeria, 206, 910, 226.4
Africa, Ethiopia, 114, 1100, 103.6
Asia, Bangladesh, 164, 130, 1261.5
Asia, India, 1380, 2973, 464.2
Asia, Japan, 126, 364, 346.2
Asia, Pakistan, 220, 770, 285.7
Asia, Indonesia, 273, 1811, 150.7
Asia, China, 1411, 9388, 150.3
Europe, Russia, 145, 16376, 8.9
North America, Mexico, 128, 1943, 65.9
North America, USA, 331, 9147, 36.2
South America, Brazil, 212, 8358, 25.4
```

We wrote prep1 to print directly into the Unix sort command, using the pipe or | operator. Each output line is piped to the command, and at the end, the sorted output is produced. See Appendix A.4.5 for more details.

The -t, argument tells sort to use a comma as its field separator. Key specifiers are interpreted in order, so the -k1, 1 argument makes the first field the primary sort key, and -k5rn argument makes the fifth field, in reverse numeric order, the secondary sort key. The secondary key is used for comparisons when the first fields are identical. (In Section 7.3, we will show a sort-generator program that creates these lists of options from a description in words.)

As an alternative, we could separate the sort process into a separate command invocation, rather than burying it within an Awk program. Print into a file with print >*file*; the file can be sorted in a separate step, as in

```
$ awk '...' >temp
$ sort temp
```

This applies to all the examples in this chapter.

We have completed the preparation and sort steps; now we need to format this information into the desired report. The program form1 does the job:

```
# form1 - format countries data by continent, pop density
BEGIN { FS = ","
    printf("%-15s %-10s %10s %7s %12s\n\n",
        "CONTINENT", "COUNTRY", "POPULATION",
        "AREA", "POP. DEN.")
    }
    { printf("%-15s %-10s %7d %10d %10.1f\n",
        $1, $2, $3, $4, $5)
}
```

The desired report can be generated by typing the command line

awk -f prep1 countries | awk -f form1

The peculiar arguments to sort in prep1 can be avoided by having the program format its output so that sort doesn't need any arguments, and then having the formatting program reformat the lines. By default, the sort command sorts its input lexicographically. In the final report, the output needs to be sorted alphabetically by continent and in reverse numerical order by population density. To avoid arguments to sort, the preparation program can put at the beginning of each line a quantity depending on continent and population density that, when sorted lexicographically, will automatically order the output correctly. One possibility is a fixed-width representation of the continent followed by the reciprocal of the population density, as in prep2:

```
# prep2 - prepare countries by continent, inverse pop density
BEGIN { FS = "\t" }
    { den = 1000*$3/$2
    printf("%-15s,%12.8f,%s,%d,%d,%.1f\n",
        $4, 1/den, $1, $3, $2, den) | "sort"
}
```

With the countries file as input, here is the output from prep2:

Africa	,	0.00441748,Nigeria,206,910,226.4
Africa	,	0.00964912,Ethiopia,114,1100,103.6
Asia	,	0.00079268, Bangladesh, 164, 130, 1261.5
Asia	,	0.00215435,India,1380,2973,464.2
Asia	,	0.00288889,Japan,126,364,346.2
Asia	,	0.00350000,Pakistan,220,770,285.7
Asia	,	0.00663370, Indonesia, 273, 1811, 150.7
Asia	,	0.00665344,China,1411,9388,150.3
Europe	,	0.11293793,Russia,145,16376,8.9
North America	,	0.01517969,Mexico,128,1943,65.9
North America	,	0.02763444,USA,331,9147,36.2
South America	,	0.03942453,Brazil,212,8358,25.4

The format \$-15s is wide enough for all the continent names, and \$12.8f covers a wide range of reciprocal densities. The final formatting program is like form1 but skips the new second field. The trick of manufacturing a sort key that simplifies the sorting options is quite general. We'll use it again in an indexing program in Chapter 6.

If we would like a slightly fancier report in which only the first occurrence of each continent name is printed, we can use the formatting program form2 in place of form1:

```
# form2 - format countries by continent, pop density
BEGIN { FS = ","
    printf("%-15s %-10s %10s %7s %12s\n",
        "CONTINENT", "COUNTRY", "POPULATION",
        "AREA", "POP. DEN.")
    }
    { if ($1 != prev) {
        print ""
        prev = $1
    } else {
            $1 = ""
    }
    printf("%-15s %-10s %7d %10d %10.1f\n",
        $1, $2, $3, $4, $5)
    }
```

The command line

awk -f prep1 countries | awk -f form2

generates this report:

CONTINENT	COUNTRY	POPULATION	AREA	POP. DEN.
Africa	Nigeria	206	910	226.4
	Ethiopia	114	1100	103.6
Asia	Bangladesh	164	130	1261.5
	India	1380	2973	464.2
	Japan	126	364	346.2
	Pakistan	220	770	285.7
	Indonesia	273	1811	150.7
	China	1411	9388	150.3
Europe	Russia	145	16376	8.9
North America	Mexico	128	1943	65.9
	USA	331	9147	36.2
South America	Brazil	212	8358	25.4

The formatting program form2 is a "control break" program: some extra processing is required at the beginning or end of a group of related items. The variable prev keeps track of the value of the continent field; only when it changes is the continent name printed.

Sometimes it's easier to handle control breaks by reading the entire input, then using simple indexing to deal with the breaks. That's the approach taken with this version, form2a:

```
# form2a - format countries by continent, pop density
BEGIN { FS = ","
        printf("%-15s %-10s %10s %7s %12s\n",
            "CONTINENT", "COUNTRY", "POPULATION",
            "AREA", "POP. DEN.")
      }
{ cont[NR] = $1; country[NR] = $2; pop[NR] = $3
  area[NR] = $4; den[NR] = $5
}
END {
  for (i = 1; i \le NR; i++) {
    if (cont[i] != cont[i-1])
      print ""
    c = cont[i] == cont[i-1] ? "" : cont[i]
    printf("%-15s %-10s %7d %10d %10.1f\n",
            c, country[i], pop[i], area[i], den[i])
  }
}
```

It seems to be about the same complexity, so it's not necessarily a win here, but in other settings it might well be.

As these examples suggest, precise formatting can often be handled by combining Awk programs. But it remains a tedious business to count characters and write printf statements to make everything line up properly, and it's a nightmare when something has to be changed.

We suggested above the possibility of building a program to format tables. Here's a program that prints items in columns. Text items are left-justified, with enough space for the widest entry in that column. Numeric items are right-justified and then centered on the widest entry. In other words, given a header in the file header and the countries file as input it would print:

\$ awk -f	table heade.	r countries	
COUNTRY	AREA	POPULATION	CONTINENT
Russia	16376	145	Europe
China	9388	1411	Asia
USA	9147	331	North America
Brazil	8358	212	South America
India	2973	1380	Asia
Mexico	1943	128	North America
Indonesia	1811	273	Asia
Ethiopia	1100	114	Africa
Nigeria	910	206	Africa
Pakistan	770	220	Asia
Japan	364	126	Asia
Banglades	h 130	164	Asia

Here's the program:

```
# table - simple table formatter
BEGIN {
    FS = "\t"; blanks = sprintf("%100s", " ")
    num_re = "^{[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$"
}
    row[NR] = $0
{
    for (i = 1; i <= NF; i++) {
        if ($i ~ num_re)
            nwid[i] = max(nwid[i], length($i))
        wid[i] = max(wid[i], length($i))
    }
}
END {
    for (r = 1; r \le NR; r++) {
        n = split(row[r], d)
        for (i = 1; i \le n; i++) {
            sep = (i < n) ? " " : "\n"
            if (d[i] ~ num_re)
                printf("%*s%s", wid[i], numjust(i,d[i]), sep)
            else
                printf("%-*s%s", wid[i], d[i], sep)
        }
    }
}
function max(x, y) \{ return (x > y) ? x : y \}
function numjust(n, s) { # position s in field n
    return s substr(blanks, 1, int((wid[n]-nwid[n])/2))
}
```

The first pass records the data and computes the maximum widths of the numeric and nonnumeric items for each column. The second pass (in the END action) prints each item in the proper position. Left-justifying alphabetic items is easy: we use wid[i], the maximum width of column i, to set the width of the format string for printf; if the maximum width is 10, for instance, the format will be \$-10s for each alphabetic item in column i. With printf, an asterisk * in a field specification is replaced by the numeric value of the next argument, so in the line

the * is replaced by the value of wid[i].

It's a bit more work for numeric items: a numeric item v in column i has to be right-justified like this:

wid[i]					
nwid[i]					
1		v			

The number of spaces to the right of v is (wid[i]-nwid[i])/2, so numjust concatenates that many spaces to the end of v, then prints it with %10s (again assuming a width of 10 characters).

Exercise 5-1. The table formatter assumes that all numbers have the same number of digits after the decimal point. Modify it to work properly if this assumption is not true. \Box

5.2 Packaged Queries and Reports

When a query is asked repeatedly, it makes sense to package it as a command that can be invoked without much typing. Suppose we want to determine the population, area, and population density of various countries. To determine this information for India, for example, we could type the command

```
awk '
BEGIN { FS = "\t" }
$1 ~ /India/ {
    printf("%s:\n", $1)
    printf("\t%d million people\n", $3)
    printf("\t%.3f million sq. km.\n", $2/1000)
    printf("\t%.1f people per sq. km.\n", 1000*$3/$2)
}
' countries
```

and get the response

```
India:
1380 million people
2.973 million sq. km.
464.2 people per sq. km.
```

Now, if we want to invoke this same command on different countries, we would get tired of substituting the new country name into the Awk program every time we executed the command. We would find it more convenient to put the program into an executable file, say info, and answer queries by typing

\$ info India
\$ info USA
...

The easiest way to pass the country name into the program is to use the -v argument, which lets us set a variable on the command line before the program is run. (This is described further in Section A.5.5 of the reference manual.)

```
awk -v country=$1 '
# info - print information about country
# usage: info country-name
BEGIN { FS = "\t" }
$1 ~ country {
    printf("\s:\n", $1)
    printf("\t%d million people\n", $3)
    printf("\t%.3f million sq. km.\n", $2/1000)
    printf("\t%.1f people per sq. km.\n", 1000*$3/$2)
}
' countries
```

This sets the variable country from the first argument when info is invoked:

Notice that any regular expression can be passed to info; in particular, it is possible to retrieve information by specifying only a part of a country name or by specifying several countries at once, as in

```
$ info 'China|USA'
```

Form Letters

Awk can be used to generate form letters by substituting values for parameters in the text of a form letter:



The text of the form letter is stored in the file letter.txt. The text contains parameters that will be replaced by a set of parameter values for each form letter that is generated. For example, the following text uses parameters #1 through #4, which represent the name of a country, and its population, area, and population density:

Subject: Demographic Information About #1 From: AWK Demographics, Inc.

In response to your request for information about #1, our latest research has revealed that its population is #2 million people and its area is #3 million square kilometers. This gives #1 a population density of #4 people per square kilometer.

From the input values

Bangladesh, 164, 0.130, 1261.5

this form letter is generated:

Subject: Demographic Information About Bangladesh From: AWK Demographics, Inc.

In response to your request for information about Bangladesh, our latest research has revealed that its population is 164 million people and its area is 0.130 million square kilometers. This gives Bangladesh a population density of 1261.5 people per square kilometer.

The program form.gen is the form-letter generator:

```
# form.gen - generate form letters
# input: prototype file letter.txt; data lines
#
   output: one form letter per data line
BEGIN {
   FS = ","
   while (getline <"letter.txt" > 0) # read form letter
        form[++n] = $0
}
   for (i = 1; i <= n; i++) { \# read data lines
{
        temp = form[i]
                             # each line generates a letter
        for (j = 1; j <= NF; j++)</pre>
            gsub("#" j, $j, temp)
       print temp
    }
}
```

The BEGIN action of form.gen reads the form-letter text from the file letter.txt and stores it in the array form; the remaining action reads the input values and uses gsub to substitute these input values in place of the parameters #n in a copy of the stored form letter. Notice how string concatenation of # and j is used to create the first argument of gsub.

5.3 A Relational Database System

In this section, we will describe a simple relational database system centered around an Awk-like query language called q, a data dictionary called the relfile, and a query processor called qawk that translates q queries into Awk programs. This system extends Awk as a database language in three ways:

Fields are referred to by name rather than by number.

The database can be spread over several files rather than just one.

A sequence of queries can be made interactively.

The advantage of symbolic rather than numeric references to fields is clear — \$area is more natural than \$2 — but the advantage of storing a database in several files may not be as obvious. A multifile database is easier to maintain, primarily because it is easier to edit a file with a small number of fields than one that contains all of them. Also, with the database system of this section it is possible to restructure the database without having to change the programs that access it. On the other hand, we have to be careful to change all relevant files whenever we add information to the database, so that it remains consistent.

Up to this point, our database has consisted of a single file named countries in which each line has four fields, named country, area, population, and continent. Suppose we add to this database a second file called capitals where each entry contains the name of a country and its capital city:

Russia	Moscow		
China	Beijing		
USA	Washington		
Brazil	Brasilia		
India	New Delhi		
Mexico	Mexico City		
Japan	Tokyo		
Ethiopia	Addis Ababa		
Indonesia	Jakarta		
Pakistan	Islamabad		
Bangladesh	Dhaka		

As in the countries file, a tab has been used to separate the fields.

From these two files, if we want to print the names of the countries in Asia along with their populations and capitals, we would have to scan both files and then piece together the results. For example, this command would work if there is not too much input data:

```
awk ' BEGIN { FS = "\t" }
    FILENAME == "capitals" {
        cap[$1] = $2
    }
    FILENAME == "countries" && $4 == "Asia" {
        print $1, $3, cap[$1]
    }
' capitals countries
```

It would certainly be easier if we could just say something like

\$continent ~ /Asia/ { print \$country, \$population, \$capital }

and have a program figure out where the fields are and how to put them together. This is how we would phrase this query in q, the language that we will describe shortly.

Natural Joins

It's time for a bit of terminology. In relational databases, a file is called a *table* or *relation* and the columns are called *attributes*. So we would say that the capitals table has the

attributes country and capital.

A *natural join*, or join for short, is an operator that combines two tables into one on the basis of their common attributes. The attributes of the resulting table are all the attributes of the two tables being joined, with duplicates removed. If we join the two tables countries and capitals, we get a single table, let's call it cc, that has the attributes

country, area, population, continent, capital

For each country that appears in both tables, we get a row in the cc table that has the name of the country, followed by its area, population, continent, and then its capital:

Russia	16376	145	Europe	Moscow
China	9388	1776	Asia	Beijing
USA	9147	331	North America	Washington
Brazil	8358	212	South America	Brasilia
India	2973	1380	Asia	New Delhi
Mexico	1943	128	North America	Mexico City
Indonesia	1811	273	Asia	Jakarta
Ethiopia	1100	114	Africa	Addis Ababa
Pakistan	770	220	Asia	Islamabad
Japan	364	126	Asia	Tokyo
Bangladesh	130	164	Asia	Dhaka

The way we implement the join operator is to sort the operand tables on their common attributes and then merge the rows if their values agree on the common attributes, as in the table above. To answer a query involving attributes from several tables, we will first join the tables, create a temporary file if necessary, and then apply the query to the resulting table. Thus to answer the query

```
$continent ~ /Asia/ { print $country, $population, $capital }
```

we join the countries and capitals tables and apply the query to the result. The trick is how, in general, to decide which tables to join.

The actual joining operation can be done by the Unix command join, but if you don't have that available, here is a basic version in Awk. It joins two files on the attribute in the first field of each. Notice that the join of the two tables

ATT1	ATT2	ATT3	 ATT1	ATT4
A	W	р	A	1
В	х	q	A	2
В	У	r	В	3
С	z	s		

is the table

ATT1	ATT2	ATT3	ATT4
A	W	р	1
A	W	р	2
В	х	q	3
В	У	r	3

In other words, join does not assume that the input tables are equally long, just that they are sorted. It makes an output line for each possible pairing of matching input fields.

```
# join - join file1 file2 on first field
   input: two sorted files, tab-separated fields
#
   output: natural join of lines with common first field
BEGIN {
   OFS = sep = " \setminus t"
    file2 = ARGV[2]
   ARGV[2] = "" # read file1 implicitly, file2 explicitly
    eofstat = 1 # end of file status for file2
    if ((ng = getgroup()) <= 0) # ng is the next group</pre>
                 # file2 is empty
        exit
}
{
   while (prefix($0) > prefix(gp[1]))
        if ((nq = qetqroup()) <= 0)
            exit # file2 exhausted
    if (prefix($0) == prefix(qp[1])) # 1st attributes in file1
        for (i = 1; i <= ng; i++)
                                     #
                                            and file2 match
            print $0, suffix(gp[i]) # print joined line
}
function getgroup() { # put equal prefix group into gp[1..ng]
    if (qetone(file2, qp, 1) \le 0)
                                     # end of file
        return 0
    for (ng = 2; getone(file2, gp, ng) > 0; ng++) {
        if (prefix(gp[ng]) != prefix(gp[1])) {
            unget(gp[ng])
                           # went too far
            return ng-1
        }
    }
    return ng-1
}
function getone(f, gp, n) { # get next line in gp[n]
    if (eofstat <= 0) # eof or error has occurred
        return 0
    if (ungot) {
                      # return lookahead line if it exists
       gp[n] = ungotline
       ungot = 0
        return 1
    }
    return eofstat = (getline gp[n] <f)</pre>
}
function unget(s) { ungotline = s; ungot = 1 }
function prefix(s) { return substr(s, 1, index(s, sep) - 1) }
function suffix(s) { return substr(s, index(s, sep) + 1) }
```

The program is called with two arguments, the two input files. Groups of lines with a common first attribute value are read from the second file. If the prefix of the line read from the first file matches the common attribute value of some group, each line of the group gives rise to a joined output line.

The function getgroup puts the next group of lines with a common prefix into the array gp; it calls getone to get each line, and unget to put a line back if it is not part of the group. We have localized the extraction of the first attribute value into the function prefix so it's easy to change.

You should examine the way in which the functions getone and unget implement a pushback or "un-read" of an input line. Before reading a new line, getone checks to see if there is a line that has already been read and stored by unget, and if there is, returns that instead of reading a new one.

Pushback is a different way of dealing with a problem that we encountered earlier, reading one too many inputs. In the control-break programs early in this chapter, we delayed processing; here we pretend, through the pair of functions getone and unget, that we never even saw the extra input.

Exercise 5-2. This version of join does not check for errors or whether the files are sorted. Remedy these defects. How much bigger is the program? \Box

Exercise 5-3. Implement a version of join that reads one file entirely into memory, then does the join. Which is simpler? \Box

Exercise 5-4. Modify join so it can join on any field or fields of the input files, and output any selected subset of fields in any order. \Box

The relfile

In order to ask questions about a database scattered over several tables, we need to know what is contained in each table. We store this information in a file called the relfile ("rel" is for relation). The relfile contains the names of the tables in the database, the attributes they contain, and the rules for constructing a table if it does not exist. The relfile is a sequence of table descriptors of the form

tablename : attribute attribute ... ! command ...

The tablenames and attributes are strings of letters. After the tablename comes a list of the names of the attributes for that table, each prefixed by spaces or tabs. Following the attributes is an optional sequence of commands prefixed by exclamation points that tell how to construct this table. If a table has no commands, a file with that name containing the data of that table is assumed to exist already. Such a table is called a *base* table. Data is entered and updated in the base tables.

A table with a sequence of commands appearing after its name in the relfile is a *derived* table. Derived tables are constructed when they are needed.

We will use the following relfile for our expanded countries database:

```
countries:
            country
            area
            population
            continent
capitals:
            country
            capital
cc:
            country
            area
            population
            continent
            capital
            !sort countries >temp.countries
            !sort capitals >temp.capitals
            !join temp.countries temp.capitals >cc
```

This file says that there are two base tables, countries and capitals, and one derived table cc that is constructed by sorting the base tables into temporary files, then joining them. That is, cc is constructed by executing

```
sort countries >temp.countries
sort capitals >temp.capitals
join temp.countries temp.capitals >cc
```

A relfile often includes a *universal relation*, a table that contains all the attributes, as the last table in the relfile. This ensures that there is one table that contains any combination of attributes. The table cc is a universal relation for the countries-capitals database.

A good design for a complex database should take into account the kinds of queries that are likely to be asked and the dependencies that exist among the attributes, but the small databases for which q is likely to be fast enough, with only a few tables, are unlikely to uncover subtleties in relfile design.

q, an Awk-like Query Language

Our query language q consists of single-line Awk programs with attribute names in place of field names. The query processor qawk answers a query as follows:

It determines the set of attributes in the query.

Starting from the beginning of the relfile, it finds the first table whose attributes include all the attributes in the query. If this table is a base table, it uses that table as the input for the query. If the table is a derived table, it constructs the derived table and uses it as the input. (This means that every combination of attributes that might appear in a query must also appear in either a base or derived table in the relfile.)

It transforms the q query into an Awk program by replacing the symbolic field references with the appropriate numeric field references. This program is then applied to the table determined in step (2).

The q query

```
$continent ~ /Asia/ { print $country, $population }
```

mentions the attributes continent, country, and population, all of which are included in the attributes of the first table countries. The query processor translates this query into the program

\$4 ~ /Asia/ { print \$1, \$3 }

which it applies to the countries file.

The q query

{ print \$country, \$population, \$capital }

contains the attributes country, population, and capital, all of which are included only in the derived table cc. The query processor therefore constructs the derived table cc using the commands listed in the relfile and translates this query into the program

{ print \$1, \$3, \$5 }

which it applies to the freshly constructed cc file.

We have been using the word "query," but it's certainly possible to use qawk to compute as well, as in this computation of the average area:

{ area += \$area }; END { print area/NR }



Figure 5-1: Behavior of qawk

qawk, a q-to-Awk Translator

We conclude this chapter with the implementation of qawk, the processor that translates q queries into Awk programs.

First, qawk reads the relfile and collects the table names into the array relname. It collects any commands needed to construct the *i*-th table and stores them into the array cmd beginning at location cmd[i, 1]. It also collects the attributes of each table into the two-dimensional array attr; the entry attr[*i*, *a*] holds the index of the attribute named *a* in the *i*-th table.

Second, qawk reads a query and determines which attributes it uses; these are all the strings of the form \$*name* in the query. Using the subset function, it determines T_i , the first table whose attributes include all of the attributes present in the query. It substitutes the indexes of these attributes into the original query to generate an Awk program, issues whatever commands are needed to create T_i , then executes the newly generated Awk program with T_i as input.

The second step is repeated for each subsequent query. The diagram in Figure 5-1 above outlines the behavior of qawk.

Here is the implementation of qawk:

```
# gawk - awk relational database guery processor
BEGIN { readrel("relfile") }
1.1
    { doquery($0) }
function doquery(s, i,j) {
   delete gattr # clean up for next guery
   query = s # put $names in query into qattr, without $
   while (match(s, /\ (A-Za-z)+/)) {
       qattr[substr(s, RSTART+1, RLENGTH-1)] = 1
       s = substr(s, RSTART+RLENGTH+1)
    }
   for (i = 1; i <= nrel && !subset(gattr, attr, i); )</pre>
       i++
   missing(qattr)
                    # table i contains attributes in query
    } else {
       for (j in qattr) # create awk program
           gsub("\\$" j, "$" attr[i,j], query)
       for (j = 1; j <= ncmd[i]; j++) # create table i</pre>
           if (system(cmd[i, j]) != 0) {
              print "command failed, query skipped\n", cmd[i,j]
              return
          }
       awkcmd = sprintf("awk -F'\t' '%s' %s", query, relname[i])
       printf("query: %s\n", awkcmd)  # for debugging
       system (awkcmd)
   }
}
function readrel(f) {
   while (getline <f > 0) { # parse relfile
       gsub(/[^A-Za-z]+/, "", $0) # remove all but name
           relname[++nrel] = $0
       } else if ($0 ~ /^[ \t]*!/)
                                    # !command...
           cmd[nrel, ++ncmd[nrel]] = substr($0, index($0, "!")+1)
       else if (0 \sim /^{[]} |A-Za-z]+[]  # attribute
           attr[nrel, $1] = ++nattr[nrel]
                                    # not white space
       else if ($0 !~ /^[ \t]*$/)
           print "bad line in relfile:", $0
   }
}
```

```
function subset(q, a, r, i) { # is q a subset of a[r]?
   for (i in q)
        if (!((r,i) in a))
            return 0
   return 1
}
function missing(x, i) {
   print "no table contains all of the following attributes:"
   for (i in x)
        print i
}
```

Exercise 5-5. If your operating system doesn't support Awk's system function, modify qawk to write the appropriate sequence of commands in a file or files that can be executed separately. \Box

Exercise 5-6. As it constructs a derived table, qawk calls system once for each command. Modify qawk to collect all of the commands for building a table into one string and to execute them with a single call to system. \Box

Exercise 5-7. Modify qawk to check whether a derived file that is going to be used as input has already been computed. If this file has been computed and the base files from which it was derived have not been modified since, then we can use the derived file without recomputing it. Look at the program make presented in Chapter 7. \Box

Exercise 5-8. Provide a way to enter and edit multiline queries. Multiline queries can be collected with minimal changes to qawk. One possibility for editing is a way to invoke your favorite text editor; another is to write a simple editor in Awk itself. \Box

5.4 Summary

In this chapter we have tried to illustrate how to use Awk to access and print information in an organized fashion, in contrast to the more typical *ad hoc* uses of earlier chapters.

For generating reports, a "divide-and-conquer" strategy is often best: prepare the data in one program, sort if necessary, then format with a second program. Control breaks can be handled either by looking behind, or, sometimes more elegantly, by an input pushback mechanism. (They can also sometimes be done by a pipeline too, although we didn't show that in this chapter.) For the details of formatting, a good alternative to counting characters by hand is to use a program that does all the mechanical parts.

Although Awk is not a tool for production databases, it is effective for small personal databases, and it also serves well for illustrating some of the fundamental notions. The qawk processor demonstrates both of these aspects. This page intentionally left blank

6

Processing Words

The programs in this chapter share a common theme: the manipulation of natural language text. The examples include programs that generate random words and sentences, that carry on limited dialogues with the user, and that process text. Most are toys, of value mainly as illustrations, but some of the document preparation programs are in regular use.

6.1 Random Text Generation

Programs that generate random data have many uses. Such programs can be created using the built-in function rand, which returns a pseudo-random number each time it is called. The rand function starts generating random numbers from the same seed each time a program using it is invoked, so if you want a different sequence each time, you must call srand (n) once, which will initialize rand with the seed n; if no seed is provided, one is computed from the current time of day. The srand function returns the previous seed so you can recreate a sequence.

Random Choices

Each time it is called, rand returns a random floating point number greater than or equal to 0 and less than 1, but often what is wanted is a random integer between 1 and n. That's easy to compute from rand:

```
# randint - return random integer k, 1 <= k <= n
function randint(n) {
    return int(n * rand()) + 1
}</pre>
```

randint (n) scales the floating point number produced by rand so it is at least 0 and less than n, truncates the fractional part to make an integer between 0 and n-1, then adds 1.

We can use randint to select random letters like this:

```
# randlet - generate random lower-case letter
function randlet() {
    return substr("abcdefghijklmnopqrstuvwxyz", randint(26), 1)
}
```

Using randint, it's also easy to print a single random element from an array of n items x[1], x[2], ..., x[n]:

```
print x[randint(n)]
```

A more interesting problem, however, is to print several random entries from the array *in the original order*. For example, if the elements of x are in increasing order, the random sample also has to be in order.

The function randk prints k random elements in order from the first n elements of an array a.

```
# randk - print in order k random elements from a[1]..a[n]
function randk(a, k, n, i) {
   for (i = 1; n > 0; i++)
        if (rand() < k/n--) {
            print a[i]
            k--
        }
}</pre>
```

In the body of the function, k is the number of entries that still need to be printed, and n is the number of array elements yet to be examined. The decision whether to print the *i*-th element is determined by the test rand() < k/n; each time an element is printed, k is decreased, and each time the test is made, n is decreased.

There's a variation on this theme: a program that will select a single random element from an input, with uniform probability regardless of how long the input is. The program randline does this:

```
# randline - print one random line of input stream
awk ' BEGIN { srand() }
        { if (rand() < 1 / ++n) out = $0 }
        END { print out }
' $*</pre>
```

It's a nice example of a useful algorithm, and it's fun (from one side at least) to use it to select students in a classroom:

```
$ randline class.list
John
$ randline class.list
Jane
$
```

One more variation, a random permutation of a set of input lines, is easily done like this:

```
BEGIN { srand() }
{ x[rand()] = $0 }
END { for (i in x) print x[i] }
```

The call to srand in the BEGIN block ensures a new permutation each time the code is run. Each input line is stored in an array with a random subscript. The lines are then printed in whatever order the for loop uses; that's also random, but fixed by the Awk implementation, so we need the rand in the second line.

Exercise 6-1. Test rand to see how random its output really is.

Exercise 6-2. The randk function above takes time proportional to n. Write a program to generate k distinct random integers between 1 and n in time proportional to k. \Box

Exercise 6-3. Write a program to generate random bridge hands.

Cliché Generation

Our next example is a cliché generator, which creates new clichés out of old ones. The input is a set of sentences like

```
A rolling stone:gathers no moss.
History:repeats itself.
He who lives by the sword:shall die by the sword.
A jack of all trades:is master of none.
Nature:abhors a vacuum.
Every man:has a price.
All's well that:ends well.
```

where a colon separates subject from predicate. Our cliché program combines a random subject with a random predicate; with luck it produces the occasional mildly amusing aphorism:

A rolling stone repeats itself. History abhors a vacuum. Nature repeats itself. All's well that gathers no moss. He who lives by the sword has a price.

The code is straightforward:

```
# cliche - generate an endless stream of cliches
# input: lines of form subject:predicate
# output: lines of random subject and random predicate
BEGIN { FS = ":" }
    { x[NR] = $1; y[NR] = $2 }
END { for (;;) print x[randint(NR)], y[randint(NR)] }
function randint(n) { return int(n * rand()) + 1 }
```

Don't forget that this program is intentionally an infinite loop.

Exercise 6-4. Modify cliche so it never prints one of the original inputs.

Random Sentences

A *context-free grammar* is a set of rules that defines how to generate or analyze a set of sentences. Each rule, called a *production*, has the form

 $A \rightarrow B \ C \ D \ \dots$

The meaning of this production is that any A can be "rewritten" as $B C D \ldots$. The symbol on the left-hand side, A, is called a *nonterminal*, because it can be expanded further. The symbols on the right-hand side can be nonterminals (including more A's) or *terminals*, so called because they do *not* get expanded. There can be several rules with the same left side; terminals and nonterminals can be repeated in right sides.

In Section 7.7 we will show a grammar for a part of Awk itself, and use that to write a parser that analyzes Awk programs. In this chapter, however, our interest is in generation, not analysis. For example, here is a grammar for sentences like "the boy walks slowly" and "the girl runs very very quickly."

```
Sentence -> Nounphrase Verbphrase
Nounphrase -> the boy
Nounphrase -> the girl
Verbphrase -> Verb Modlist Adverb
Verb -> runs
Verb -> walks
Modlist ->
Modlist ->
Modlist -> very Modlist
Adverb -> quickly
Adverb -> slowly
```

We use upper case names for nonterminals and lower case for terminals.

The productions generate sentences for nonterminals as follows. Suppose Sentence is the starting nonterminal. Choose a production with that nonterminal on the left-hand side:

Sentence -> Nounphrase Verbphrase

Next pick any nonterminal from the right side, for example, Nounphrase, and rewrite it with any one of the productions for which it is the left side:

Sentence -> Nounphrase Verbphrase -> the girl Verbphrase

Now pick another nonterminal from the resulting right side (this time only Verbphrase remains) and rewrite it by one of its productions:

Sentence -> Nounphrase Verbphrase -> the girl Verbphrase -> the girl Verb Modlist Adverb

Continue rewriting this way until no more nonterminals remain:

```
Sentence -> Nounphrase Verbphrase
-> the girl Verbphrase
-> the girl Verb Modlist Adverb
-> the girl runs very Modlist Adverb
-> the girl runs very Adverb
-> the girl runs very quickly
-> the girl runs very very quickly
```

The result is a sentence for the starting nonterminal. This derivation process is the opposite of the sentence-diagraming procedure taught in elementary school: rather than combining an adverb and a verb into a verb phrase, we are expanding a verb phrase into a verb and an adverb.

The productions for Modlist are interesting. One rule says to replace Modlist by very Modlist; each time we do this, the sentence gets longer. Fortunately, this potentially

infinite process terminates as soon as we replace Modlist by the other possibility, which is the null string.

We will now present a program to generate sentences in a grammar, starting from any specified nonterminal. The program reads the grammar from a file and records the number of times each left-hand side occurs, plus the number of right-hand sides it has, and the components of each. Thereafter, whenever a nonterminal is typed, a random sentence for that non-terminal is generated.

The data structure created by this program uses three arrays to store the grammar: lhs[A] gives the number of productions for the nonterminal A on the left-hand side, rhscnt[A,i] gives the number of symbols on the right-hand side of the i-th production for A, and rhslist[A,i,j] contains the j-th symbol in the i-th right-hand side for A. For our grammar, these arrays contain:



The complete program begins at the top of the next page.

The function gen ("A") generates a sentence for the nonterminal A. It calls itself recursively to expand nonterminals introduced by previous expansions. We must remember to make sure that all the temporary variables used by a recursive function appear in the parameter list of the function declaration, as in gen. If they do not, they are global variables, and the program won't work properly.

We chose to use separate arrays for the right-hand-side counts and components, but it is possible instead to use subscripts to encode different fields, rather like records or structures in other languages. For example, the array rhscnt[i,j] could be part of rhslist, as rhslist[i,j,"cnt"].

Exercise 6-5. Write a grammar for generating plausible-sounding text from a field that appeals to you — business, politics, and computing are all good possibilities. \Box

Exercise 6-6. With some grammars, there is an unacceptably high probability that the sentence-generation program will go into a derivation that just keeps getting longer. Add a mechanism to limit the length of a derivation. \Box

Exercise 6-7. Add probabilities to the rules of a grammar, so that some of the rules associated with a nonterminal are more likely to be chosen than others. \Box

```
# sentgen - random sentence generator
#
   input: grammar file; sequence of nonterminals
#
   output: a random sentence for each nonterminal
BEGIN { # read rules from grammar file
    while (getline < "grammar" > 0)
        if ($2 == "->") {
            i = ++lhs[\$1]
                                       # count lhs
            rhscnt[$1, i] = NF-2 # how many in rhs
            for (j = 3; j \le NF; j++) # record them
               rhslist[$1, i, j-2] = $j
        } else
           print "illegal production: " $0
}
{
   if ($1 in lhs) { # nonterminal to expand
        gen($1)
        printf("\n")
    } else
       print "unknown nonterminal: " $0
}
function gen(sym, i, j) { # print random phrase derived from sym
    if (sym in lhs) {
                            # a nonterminal
        i = int(lhs[sym] * rand()) + 1
                                        # random production
        for (j = 1; j <= rhscnt[sym, i]; j++) # expand rhs's</pre>
            gen(rhslist[sym, i, j])
    } else
        printf("%s ", sym)
}
```

Exercise 6-8. Implement a nonrecursive version of the sentence-generation program. \Box

6.2 Interactive Text-Manipulation

Awk can be used to write interactive programs. We'll illustrate the basic ideas with two programs. The first tests arithmetic skills, and the second tests knowledge of particular subject areas.

Skills Testing: Arithmetic

The following program arith (best suited for a very young child) presents a sequence of addition problems like

7 + 9 = ?

After each problem, the user types an answer. If the answer is right, the user is praised and presented with another problem. If the answer is wrong, the program asks for the answer again. If the user provides no answer at all, the right answer is printed before the next problem is presented.

The program is invoked with one of two command lines, either this one:

\$ awk -f arith

or this one:

\$ awk -f arith maxnum

If there is an argument after arith on the command line, the argument is used to limit the maximum size of the numbers in each problem. After this argument has been read, ARGV[1] is reset to "-" so the program will be able to read the answers from the standard input. If no argument is specified, the maximum size will be 10.

```
# arith - addition drill
#
    usage: awk -f arith [ optional problem size ]
#
    output: queries of the form "i + j = ?"
BEGIN {
    maxnum = ARGC > 1 ? ARGV[1] : 10
                                      # default size is 10
    ARGV[1] = "-" # read standard input subsequently
                   # reset rand from time of day
    srand()
    do {
        n1 = randint(maxnum)
        n2 = randint(maxnum)
        printf("%g + %g = ? ", n1, n2)
        while ((input = getline) > 0) {
            if (\$0 == n1 + n2) {
                print "Right!"
                break
            } else if ($0 == "") {
                print n1 + n2
                break
            } else
                printf("wrong, try again: ")
        }
    } while (input > 0)
}
function randint(n) {
    return int(rand()*n)+1
}
```

Exercise 6-9. Add the other arithmetic operators. *Exercise 6-10.* Add a way to provide hints for wrong answers.

Skills Testing: Quiz

Our second example is a program called quiz that asks questions from some specified file of questions and answers. For example, consider testing knowledge of chemical elements. Suppose the question-and-answer file quiz.elems contains the symbol, atomic number, and full name for each element, separated by colons. The first line identifies the fields of subsequent lines, with alternatives separated by vertical bars:

```
symbol:number:name|element
H:1:Hydrogen
He:2:Helium
Li:3:Lithium
Be:4:Beryllium
B:5:Boron
C:6:Carbon
N:7:Nitrogen
O:8:Oxygen
F:9:Fluorine
Ne:10:Neon
Na:11:Sodium|Natrium
...
```

The quiz program is shown in Figure 6-1. It uses the first line to decide which field is the question and which is the answer, then reads the rest of the file into an array, from which it presents random items and checks answers. After typing the command line

\$ awk -f quiz quiz.elems name symbol

we might engage in a dialogue like this:

```
Beryllium? B
wrong, try again: Be
Right!
Fluorine?
```

Notice that alternative answers (for example, sodium or natrium) are easily handled with regular expressions in the data file. We have to surround the regular expression for the right answer with \uparrow and \$; without this, any matching substring of the right answer would also be accepted (so N would match Ne and Na as well as N).

We have also arranged that the error messages are printed on /dev/stderr, just in case the program's standard output is being directed to a file.

Exercise 6-11. Modify quiz so that it does not present any question more than once.

6.3 Text Processing

Because of its string manipulation capabilities, Awk is useful for tasks that arise in text processing and document preparation. As examples, this section contains programs for counting words, formatting text, maintaining cross-references, making KWIC indexes, and preparing indexes.

Word Counts

In Chapter 1, we presented a program to count the number of lines, words, and characters in a file, where a word was defined as any contiguous sequence of nonspace, nontab characters.

A related problem is to count the number of times each different word appears in a document. One way to solve this problem is to isolate the words, sort them to bring identical words together, and then count occurrences of each word with a control-break program.

```
# quiz - present a quiz
#
  usage: awk -f quiz topicfile question-subj answer-subj
BEGIN {
   FS = ":"
   if (ARGC != 4)
       error("usage: awk -f quiz topicfile question answer")
    if (getline <ARGV[1] < 0) # 1st line is subj:subj:...
       error("no such quiz as " ARGV[1])
    for (q = 1; q \le NF; q++)
        if ($q ~ ARGV[2])
           break
    for (a = 1; a \le NF; a++)
        if ($a ~ ARGV[3])
           break
    if (q > NF || a > NF || q == a)
        error("valid subjects are " $0)
    while (getline <ARGV[1] > 0) # load the quiz
        qa[++nq] = $0
   ARGC = 2; ARGV[1] = "-"  # now read standard input
    srand()
    do {
        split(ga[int(rand()*ng + 1)], x)
        printf("%s? ", x[q])
       while ((inputstat = getline) > 0) {
            if ($0 ~ "^(" x[a] ")$") {
                print "Right!"
                break
            } else if ($0 == "") {
                print x[a]
                break
            } else {
                printf("wrong, try again: ")
            }
        }
    } while (inputstat > 0)
}
function error(s) {
   printf("error: %s\n", s) > "/dev/stderr"
   exit
}
```

Figure 6-1: Implementation of quiz program

Another way, well suited to Awk, is to isolate the words and aggregate the count for each word in an associative array. To do this properly, we have to decide what a word really is. In the following program, a word is a field with the punctuation removed, so that, for example, "word" and "word;" and "(word)" are all counted in the entry for word. The END action prints the word frequencies, sorted into decreasing order.
```
# wordfreq - print number of occurrences of each word
# input: text
# output: number-word pairs sorted by number
{ gsub(/[.,:;!?(){}]/, "") # remove punctuation
  for (i = 1; i <= NF; i++)
        count[$i]++
  }
END { for (w in count)
        print count[w], w | "sort -rn"
        close("sort -rn")
  }
```

Here are the top dozen words for a draft of this book:

3378	the	1696	of	1574	а	1363	is	1254	to	1222	and
969	in	659	The	621	that	533	are	517	program	507	for

It would easy to combine upper case words like The with lower case like the by using the tolower function as the counts are being accumulated, but this discards information about case, which might be relevant. If we sort in a case-independent manner with sort -fd, that will bring together words spelled the same way except for case, while ignoring nonalphabetic characters. This can reveal potential problems like inconsistent hyphenation (commandline vs command-line) or capitalization (JavaScript vs Javascript) or spelling (judgment vs judgement).

This might also be a good place to use one of the shorthands for character classes that are Unicode-aware. For example, in a regular expression, [[:punct:]] matches a single punctuation character in the local character set, as defined by the locale shell variable. It is a drop-in replacement for the list of punctuation characters in the program that will work properly in the appropriate local language.

There are similar notations for digits, alphanumeric characters, white space, and so on, described in Section A.1.4 of the reference manual. We have not used them in the body of the text very much, but for programs that are meant to work in environments other than English-speaking North America, they can be a better choice.

As an alternative to sort arguments, you can prefix each line by a sort key that has the right properties for the desired sort. For example, a sequence like

```
pfx = tolower($0)
gsub(/[^A-Za-z]/, "", pfx)
print pfx, $0
```

creates a lower-case-only sort prefix with all non-alphabetic characters removed, equivalent to a dictionary-order sort. This is another place where a Unicode-aware regular expression like / [[:alpha:]] / would make the test more reliable in non-ASCII environments.

Exercise 6-12. Modify wordfreq to exclude "stop words" like the common ones listed above. *Exercise 6-13.* Modify wordfreq to focus on very specific kinds of words, perhaps weasel-words and probably quite unnecessary adverbs like "quite," "probably," "perhaps" and "very."

Exercise 6-14. Write a program to count the number of sentences in a document and their lengths. \Box

Text Formatting

Our next example, fmt, formats its input into lines that are at most 60 characters long, by moving words to fill each line as much as possible. Blank lines cause paragraph breaks; otherwise, there are no commands. It's useful for formatting text that was originally created without thought to line length.

```
# fmt - format text into 60-char lines
/./ { for (i = 1; i <= NF; i++) addword($i) }
/^$/ { printline(); print "" }
END { printline() }
function addword(w) {
    if (length(line) + length(w) > 60)
        printline()
    line = line space w
        space = " "
}
function printline() {
    if (length(line) > 0)
        print line
    line = space = "" # reset for next line
}
```

In a sense, fmt is the barest minimum version of Markdown, a widely-used language for formatting text without using explicit formatting commands. Of all the programs in this book, it's the most frequently used by (one of) the authors.

Exercise 6-15. Modify fmt so that the line length can be set by a numeric command-line argument, or by using -v. \Box

Exercise 6-16. Modify fmt to align the right margin of the text it prints by adding extra spaces within the line. \Box

Exercise 6-17. Enhance fmt to infer the proper format of a document by recognizing probable titles, headings, lists, and other facilities provided by Markdown. Rather than formatting, it could generate formatting commands for subsequent formatting with troff, LaTeX or HTML. \Box

Maintaining Cross-References in Manuscripts

A common document preparation problem is to create consistent names or numbers for items like bibliographic citations, figures, tables, examples, and so on. Some text formatters help with this task, but most expect you to do it yourself. Our next example is a technique for numbering cross-references. It's useful for documents like technical papers or books.

As the document is being written, the author creates and uses symbolic names for the various items that will be cross-referenced. Because the names are symbolic, items can be added, deleted, and rearranged without having to change any existing names. Two programs create the version in which the symbolic names are replaced by suitable numbers. Here is a sample document containing symbolic names for three bibliographic citations and one figure:

```
.#Fig _quotes_
Figure _quotes_ gives three brief quotations from famous books.
                        Figure _quotes_:
.#Bib _alice_
  "... 'and what is the use of a book, ' thought Alice,
  'without pictures or conversations?'" [_alice_]
.#Bib _huck_
  "... if I'd a knowed what a trouble it was to make a book
  I wouldn't a tackled it and ain't agoing to no more." [_huck_]
.#Bib _bible_
  "... of making many books there is no end; and much study
  is a weariness of the flesh." [_bible_]
[_alice_] Carroll, L., Alice's Adventures in Wonderland,
    Macmillan, 1865.
[_huck_] Twain, M., Adventures of Huckleberry Finn,
   Webster & Co., 1885.
[_bible_] King James Bible, Ecclesiastes 12:12.
```

Each symbolic name is defined by a line of the form

.#Category _SymbolicName_

Such a definition can appear anywhere in the document, and there can be as many different categories as the author wants. Throughout the document an item is referred to by its symbolic name. We have chosen symbolic names that begin and end with an underscore, but any names can be used as long as they can be separated from other text. (Item names must all be distinct, even if in different categories; this simplifies the code.) The names .#Fig and .#Bib begin with a period so they will be ignored by the troff formatter in case the document is printed without resolving the cross-references; with a different formatter, a different convention may be required.

The conversion creates a new version of the document in which the definitions are removed and each symbolic name is replaced by a number. In each category the numbers start at one and go up sequentially in the order in which the definitions for that category appear in the original document.

The conversion is done by passing the document through two programs. This division of labor is another instance of a powerful general technique: the first program creates a second program to do the rest of the job; it's a program that writes a program. In this case, the first program, called xref, scans the document and creates the second program, called xref.conv, that does the actual conversion.

If the original version of the manuscript is in the file document, the version with the numeric references is created by typing:

```
$ awk -f xref document >xref.conv
$ awk -f xref.conv document
```

The output of the second program can be directed to a printer or text formatter. The result for our sample above is:

[3] King James Bible, Ecclesiastes 12:12.

The xref program searches the document for lines beginning with ". #"; for each such definition it increments a counter in the array count for items of that category and prints a gsub statement.

```
# xref - create numeric values for symbolic names
# input: text with definitions for symbolic names
# output: awk program to replace symbolic names by numbers
/^\.#/ { printf("{ gsub(/%s/, \"%d\") }\n", $2, ++count[$1]) }
END { printf("!/^[.]#/\n") }
```

The output of xref on the file above is the second program, xref.conv:

```
{ gsub(/_quotes_/, "1") }
{ gsub(/_alice_/, "1") }
{ gsub(/_huck_/, "2") }
{ gsub(/_bible_/, "3") }
!/^[.]#/
```

The gsub functions globally substitute numbers for the symbolic names; the last statement deletes the definitions by not printing lines that begin with .#.

Exercise 6-18. What might happen if the trailing underscore were omitted from a symbolic name? \Box

Exercise 6-19. Modify xref to detect multiple definitions of a symbolic name. \Box

Exercise 6-20. Modify xref to create editing commands for your favorite text or stream editor (e.g., sed) instead of creating Awk commands. What effect does this have on performance? \Box

Exercise 6-21. How could you modify xref to make only a single pass over the input? What restrictions on placement of definitions does this imply? \Box

Making a KWIC Index

A Keyword-In-Context or KWIC index is an index that shows each word in the context of the line it is found in. KWIC indexes are sometimes called permuted indexes or concordances. Consider the three sentences

All's well that ends well. Nature abhors a vacuum. Every man has a price.

Here is a KWIC index for these sentences:

```
Every man has a price.

Nature abhors a vacuum.

Nature abhors a vacuum.

All's well that ends well.

All's well that ends well.

Every man has a price.

Every man has a price.

Every man has a price.

Nature abhors a vacuum.

Every man has a price.

All's well that ends well.

Nature abhors a vacuum.

All's well that ends well.

All's well that ends well.

All's well that ends well.
```

The problem of constructing a KWIC index has had an interesting history in the field of software engineering. It was proposed as a design exercise by computer scientist David Parnas in 1972; he presented a solution based on a single program. The Unix command ptx, which does the same job in much the same way, is about 500 lines of C.

The convenience of Unix pipelines suggests a three-step solution: a first program generates rotations of each input line so that each word in turn is at the front, a sort puts them in order, and another program unrotates them.

This method is even easier with Awk; it can be done by a pair of short Awk programs with a sort between them:

```
# kwic - generate kwic index
awk '
{
    print $0
    for (i = length($0); i > 0; i--) { # compute length only once
        if (substr($0,i,1) == " ")
            # prefix space suffix ==> suffix tab prefix
            print substr($0,i+1) "\t" substr($0,1,i-1)
        }
} ' $* |
sort -f |
awk '
BEGIN { FS = "\t"; WID = 30 }
        { printf("%*s %s\n", WID, substr($2,length($2)-WID+1),
            substr($1,1,WID))
} '
```

The first program prints a copy of each input line. It also prints an output line for every space within each input line; the output consists of the part of the input line after the space, followed by a tab, followed by the part before the space.

All output lines are then piped into the Unix command sort -f which sorts them, "folding" upper and lower-case letters together, so that, for example, Jack and jack will appear adjacent. (It might be useful to include the -d option as well; "dictionary" order ignores non-letters while sorting.)

From the output of the sort command, the second Awk program reconstructs the input lines, appropriately formatted. It prints a portion of the part after the tab, followed by a couple of spaces, then a portion of the part in front of the tab, all positioned so that the keywords line up.

Note the format conversion %*s: the field width is determined by the next argument in the call of printf and that value replaces the asterisk.

A KWIC or permuted index can be useful for detecting anomalies in writing, like spelling errors, because the process brings together words that share a common prefix but may differ later. A variation that works on columns in a dataset will have the same desirable properties.

Exercise 6-22. Add a list of "stop words" to kwic: a set of words like "a" and "the" that are not to be taken as keywords.

Exercise 6-23. Fix kwic to show as much as possible of lines, by wrapping around at the ends rather than truncating. \Box

Exercise 6-24. Write a program to make a concordance instead of a KWIC index: for each significant word, show all the sentences or phrases where the word appears. \Box

6.4 Making an Index

A major document like a book or a manual usually needs an index. There are three parts to indexing. The first is deciding on the terms to be indexed; this is demanding intellectual work if done well, and is hard to mechanize. The second is to insert indexing terms in the text that will capture page numbers as the text is formatted. The third part really is mechanical: producing, from a list of index terms and page numbers, a properly alphabetized and formatted index, like the one at the back of this book.

In the remainder of this section, we are going to use Awk and the sort command to build the core of an indexer (whose slightly bigger sibling was used to create the index of this book).

The basic idea comes from Jon Bentley, and is similar to the KWIC index program: divide and conquer. The job is broken down into a sequence of easy pieces, each based on a sort command or a short Awk program. Since the pieces are tiny and separate, they can easily be adapted or augmented with others to satisfy more complicated indexing requirements.

These programs contain details that are specific to the troff formatter, which we used to typeset this book. These would change if the programs were to be used with another formatter, such as LaTeX, but the basic structure would be the same. In any case, you can certainly ignore them.

We indexed the book by inserting formatting commands into the text. When the text is run through troff, these commands cause index terms and page numbers to be collected in a file. This produces a sequence of lines like the following, which is the raw material for the index-preparation programs (a single tab separates the number from the index term):

```
[FS] variable
                        35
[FS] variable
                        36
arithmetic operators
                        36
coercion rules
                        44
                       44
string comparison
numeric comparison
                        44
                       44
arithmetic operators
                       45
coercion~to number
                       45
coercion~to string
                       47
[if]-[else] statement
control-flow statements 48
[FS] variable
                        52
. . .
```

The intent is that an index term like

string comparison 44

should ultimately appear in the index in two forms:

string comparison 44 comparison, string 44

Index terms are normally split and rotated at each space in the term. The tilde \sim indicates a space that will not be split:

coercion~to number 45

is not to be indexed under "to."

There are a couple of other frills. Since we use troff, some troff size- and fontchange commands are recognized and properly ignored during sorting. Furthermore, because font changes occur frequently in the index, we use the shorthand [...] to indicate material that should appear in the index in the constant-width font; for example

[if]-[else] statement

is to be printed as

if-else statement 47 statement, if-else 47

The indexing process is a composition of six commands:

ix.sort1	sort input pairs by index term, then by page number
ix.collapse	collapse number lists for identical terms
ix.rotate	generate rotations of index term
ix.genkey	generate a sort key to force proper ordering
ix.sort2	sort by sort key
ix.format	generate final output

These commands gradually massage the index-term, page-number pairs into the final form of the index, which is eventually typeset along with the rest of the book. For the remainder of this section we will consider these commands in order.

The initial sort takes the index-term, page-number pairs as input and brings identical terms together in page-number order:

```
# ix.sort1 - sort by index term, then by page number
# input/output: lines of the form string tab number
# sort by string, then by number; discard duplicates
sort -t'tab' -k1 -k2n -u
```

The arguments to the sort command need explanation: -t'tab' says tab is the field separator; -k1 says the first sort key is field 1, which is to be sorted alphabetically; -k2n says the second sort key is field 2, which is to be sorted numerically; and -u says to discard duplicates. (In Section 7.3, we describe a sort-generator program that will create these arguments for you.) The output of *ix.sort1* on the input above is:

[FS] variable	35
[FS] variable	36
[FS] variable	52
[if]-[else] statement	47
arithmetic operators	36
arithmetic operators	44
coercion rules	44
coercion~to number	45
coercion~to string	45
control-flow statements	48
numeric comparison	44
string comparison	44

This output becomes the input to the next program, ix.collapse, which puts the page numbers for identical terms on a single line, using a variation of the usual control-break program.

```
# ix.collapse - combine number lists for identical terms
# input: string tab num \n string tab num ...
# output: string tab num num ...
BEGIN { FS = OFS = "\t" }
$1 != prev {
    if (NR > 1)
        printf("\n")
    prev = $1
    printf("\s\t\ss", $1, $2)
    next
}
    { printf(" %s", $2) }
END { if (NR > 1) printf("\n") }
```

The output of ix.collapse is

[FS] variable	35	36	52
[if]-[else] statement	47		
arithmetic operators	36	44	
coercion rules	44		
coercion~to number	45		
coercion~to string	45		
control-flow statements	48		
numeric comparison	44		
string comparison	44		

The next program, ix.rotate, produces rotations of the index terms from this output, for example generating "comparison, string" from "string comparison." This is much the same computation as in the KWIC index, although we've written it differently. Notice the assignment expression in the for loop.

```
# ix.rotate - generate rotations of index terms
#
    input: string tab num num ...
#
    output: rotations of string tab num num ...
BEGIN {
    FS = " \setminus t"
    OFS = "\t"
}
{
    print $1, $2
                   # unrotated form
    for (i = 1; (j = index(substr($1, i+1), "")) > 0; ) {
        i += j
                 # find each blank, rotate around it
        printf("%s, %s\t%s\n",
            substr($1, i+1), substr($1, 1, i-1), $2)
    }
}
```

The output from ix.rotate begins

[FS] variable	35	36	52
variable, [FS]	35	36	52
[if]-[else] statement	47		
<pre>statement, [if]-[else]</pre>	47		
arithmetic operators	36	44	
operators, arithmetic	36	44	
coercion rules	44		
rules, coercion	44		
coercion~to number	45		
number, coercion~to	45		
coercion~to string	45		
string, coercion~to	45		
control-flow statements	48		
statements, control-flow	48		
numeric comparison	44		
comparison, numeric	44		
string comparison	44		
comparison, string	44		

The next stage is to sort these rotated index terms. The problem with sorting them directly is that there may still be embedded formatting information like [...] that will interfere with the sort order. So each line is prefixed with a sort key that ensures the proper order; the sort key will be stripped off later.

The program ix.genkey creates the key from the index term by removing troff size and font change commands, which look like $\s+n$, or $\s-n$, or \fx , or \fx . It also converts the tildes to spaces, and removes any nonalphanumeric characters other than space from the sort key.

```
# ix.genkey - generate sort key to force ordering
# input: string tab num num ...
# output: sort key tab string tab num num ...
BEGIN { FS = OFS = "\t" }
{ gsub(/~/, " ", $1)  # tildes now become spaces
    key = $1
    # remove troff size and font change commands from key
    gsub(/\f.|\\f\(..|\\s[-+][0-9]/, "", key)
    # keep spaces, commas, letters, digits only
    gsub(/[^a-zA-Z0-9, ]+/, "", key)
    if (key ~ /^[^a-zA-Z]/) # force nonalpha to sort first
        key = " key # by prefixing a space
    print key, $1, $2
}
```

The output is now

FS variable	[FS] variable	35	36	52
variable, FS	variable, [FS]	35	36	52
ifelse statement	[if]-[else] statement	47		
statement, ifelse	<pre>statement, [if]-[else]</pre>	47		
arithmetic operators	arithmetic operators	36	44	
operators, arithmetic	operators, arithmetic	36	44	
coercion rules	coercion rules	44		
rules, coercion	rules, coercion	44		
coercion to number	coercion to number	45		

The first few lines should clarify the distinction between the sort key and the actual data.

The second sort puts terms into alphabetical order; as before, the -f option folds upper and lower case together, and -d is dictionary order.

```
# ix.sort2 - sort by sort key
# input/output: sort-key tab string tab num num ...
sort -f -d
```

This puts items into their final order:

arithmetic operators	arithmetic operators	36	44	
coercion rules	coercion rules	44		
coercion to number	coercion to number	45		
coercion to string	coercion to string	45		
comparison, numeric	comparison, numeric	44		
comparison, string	comparison, string	44		
controlflow statements	control-flow statements	48		
FS variable	[FS] variable	35	36	52
ifelse statement	[if]-[else] statement	47		
number, coercion to	number, coercion to	45		

The last stage, ix.format, removes the sort key, expands any [...] into font-change commands, and precedes each term by a formatting command .XX that can be used by a text formatter to control size, position, etc. (The actual command sequences are specific to troff; you can ignore the details.)

```
# ix.format - remove key, restore size and font commands
# input: sort key tab string tab num num ...
# output: troff format, ready to print
BEGIN { FS = "\t" }
{ gsub(/ /, ", ", $3)  # commas between page numbers
   gsub(/\[/, "\\f(CW", $2)  # set constant-width font
   gsub(/\]/, "\\fP", $2)  # restore previous font
   print ".XX"  # user-definable command
   printf("%s %s\n", $2, $3) # actual index entry
}
```

The final output begins like this:

```
.XX
arithmetic operators 36, 44
.XX
coercion rules 44
.XX
coercion to number 45
...
```

To recapitulate, the indexing process consists of a pipeline of six commands

```
sh ix.sort1 |
awk -f ix.collapse |
awk -f ix.rotate |
awk -f ix.genkey |
sh ix.sort2 |
awk -f ix.format
```

If these are applied to the input of index-term, page-number pairs at the beginning of this section, and formatted, the result looks like this:

arithmetic operators 36, 44 coercion rules 44 coercion to number 45 coercion to string 45 comparison, numeric 44 comparison, string 44 control-flow statements 48 FS variable 35, 36, 52 if-else statement 47 number, coercion to 45 numeric comparison 44 operators, arithmetic 36, 44 rules, coercion 44 statement, if-else 47 statements, control-flow 48 string, coercion to 45 string comparison 44 variable, FS 35, 36, 52

Many enhancements and variations are possible; some of the most useful are suggested in the exercises. The important lesson, however, is that dividing the job into a sequence of tiny programs makes the whole task simpler and easier to adapt to new requirements. *Exercise 6-25.* Modify or augment the indexing programs to provide hierarchical indexes, *See* and *See also* terms, and Roman-numeral page numbers. \Box

Exercise 6-26. Allow literal [,], ~, and % characters in index terms.

Exercise 6-27. Attack the problem of creating an index automatically by building tools that prepare lists of words, phrases, etc. How well does the list of word frequencies produced by wordfreq suggest index terms or topics? \Box

6.5 Summary

Awk programs can manipulate text with much the same ease that languages like C or Java manipulate numbers — storage is managed automatically, and the built-in operators and functions provide many of the necessary services. As a result, Awk is usually good for prototyping, and sometimes it is entirely adequate for production use. The indexing programs are a good example — we used a version of them to index this book.

This page intentionally left blank

7

Little Languages

Awk is often used to develop translators for "little languages," that is, languages for specialized applications. One reason for writing a translator is to learn how a language processor works. The first example in this chapter is an assembler that in twenty lines or so shows the essentials of the assembly process. It is accompanied by an interpreter that executes the assembled programs. The combination illustrates the rudiments of assembly language and computer architecture. Other examples show the basic operation of several calculators and of a recursive-descent translator for a subset of Awk itself.

You might want to experiment with the syntax or semantics of a special-purpose language before making a large investment in implementation. As examples, this chapter describes languages for drawing graphs and for specifying sort commands.

Or you might want to make a language for practical use, such as one of the several calculators in this chapter.

Language processors are built around this conceptual model:



The front end, the analyzer, reads the source program and breaks it apart into its lexical units: operators, operands, and so on. It parses the source program to check that it is grammatically correct, and if it is not, issues the appropriate error messages. Finally, it translates the source program into some intermediate representation from which the back end, the synthesizer, generates the target program. The symbol table communicates information collected by the analyzer about the source program to the synthesizer, which uses it during code generation. Although we have described language processing as a sequence of clearly distinguishable phases, in practice the boundaries are often blurred and the phases may be combined.

Awk is useful for creating processors for experimental languages because its basic operations support many of the tasks involved in language translation. Simple syntax analysis can be handled with field splitting and regular expression pattern matching. Symbol tables can be managed with associative arrays. Code generation can be done with printf statements.

In this chapter we will develop several translators to illustrate these points. In each case, we will do the minimum that will make the point or teach the lesson; embellishments and refinements are left as exercises.

7.1 An Assembler and Interpreter

Our first example of a language processor is an assembler for a hypothetical computer of the sort encountered in an introductory course on computer architecture or systems programming; it's loosely similar to some early minicomputers, and our inspiration and initial implementation both come from Jon Bentley. The computer has a single accumulator, ten instructions, and a word-addressable memory of 1000 words. We'll assume that a "word" of computer memory holds five decimal digits; if the word is an instruction, the first two digits encode the operation and the last three digits are the address. The assembly-language instructions are shown in Table 7-1.

OPCODE	INSTRUCTION	MEANING
01	get	read a number from the input into the accumulator
02	put	write the contents of the accumulator to the output
03	ld M	load accumulator with contents of memory location M
04	st M	store contents of accumulator in location M
05	add M	add contents of location M to accumulator
06	sub M	subtract contents of location M from accumulator
07	jpos M	jump to location M if accumulator is positive
08	jz M	jump to location M if accumulator is zero
09	јM	jump to location M
10	halt	stop execution
	const C	assembler pseudo-operation to define a constant C

TABLE 7-1. ASSEMBLY-LANGUAGE INSTRUCTIONS

An assembly-language program is a sequence of statements, each consisting of three fields: label, operation, and operand. Any field may be empty; labels must begin in column one. A program may also contain comments like those in Awk programs. An assembler converts these statements into instructions in the native format of the computer, as illustrated in this figure:

Here is a sample assembly-language program that prints the sum of a sequence of integers; the end of the input is marked by a zero.

```
# print sum of input numbers (terminated by zero)
    ld
          zero
                 # initialize sum to zero
    st.
          sum
loop get
                 # read a number
          done
                 # no more input if number is zero
     jΖ
          sum
    add
                 # add in accumulated sum
    st
          sum
                 # store new value back in sum
                # go back and read another number
    j
          loop
done ld
                 # print sum
          sum
    put
    halt
zero const 0
sum const
```

The target program resulting from translating this program into machine language is a sequence of integers that represents the contents of memory when the target program is ready to be run. For this program, the memory contents can be represented like this:

0:	03010		ld	zero	#	initialize sum to zero
1:	04011		st	sum		
2:	01000	loop	get		#	read a number
3:	08007		jz	done	#	no more input if number is zero
4:	05011		add	sum	#	add in accumulated sum
5:	04011		st	sum	#	store new value back in sum
6:	09002		j	loop	#	go back and read another number
7:	03011	done	ld	sum	#	print sum
8:	02000		put			
9:	10000		halt			
10:	00000	zero	const	0		
11:	00000	sum	const			

The first field is the memory location; the second is the encoded instruction. Memory location 0 contains the translation of the first instruction of the assembly-language program, 1d zero.

The assembler does its translation in two passes. Pass 1 uses field splitting to do lexical and syntactic analysis. It reads the assembly-language program, discards comments, assigns a memory location to each label, and writes an intermediate representation of operations and operands into a temporary file. Pass 2 reads the temporary file, converts symbolic operands to the memory locations computed by pass 1, encodes the operations and operands, and puts the resulting machine-language program into the array mem.

As the other half of the job, we'll build an interpreter that simulates the behavior of the computer on machine-language programs. The interpreter implements the classic fetch-decode-execute cycle: fetch an instruction from mem, decode it into an operator and an operand, and then simulate the instruction. The program counter is kept in the variable pc.

```
# asm - assembler and interpreter for simple computer
# usage: awk -f asm program-file data-files...
BEGIN {
   srcfile = ARGV[1]
  ARGV[1] = "" # remaining files are data
  tempfile = "asm.temp"
   n = split ("const get put ld st add sub jpos jz j halt", x)
   for (i = 1; i \le n; i++) # create table of op codes
      op[x[i]] = i-1
# ASSEMBLER PASS 1
   FS = "[ \t]+" # multiple spaces and/or tabs as separator
   while (getline <srcfile > 0) {
      sub(/#.*/, "")
                             # strip comments
      symtab[$1] = nextmem  # remember label location
if ($2 != "") {  # save op, addr if present
         print $2 "\t" $3 >tempfile
         nextmem++
      }
   }
   close(tempfile)
# ASSEMBLER PASS 2
   nextmem = 0
   while (getline <tempfile > 0) {
      if ($2 !~ /^[0-9]*$/) # if symbolic addr,
                             # replace by numeric value
         $2 = symtab[$2]
     mem[nextmem++] = 1000 * op[$1] + $2 # pack into word
   }
# INTERPRETER
   for (pc = 0; pc \ge 0;)  {
      addr = mem[pc] % 1000
      code = int(mem[pc++] / 1000) # advance pc to next instruction
     if
            (code == op["get"]) { getline acc }
     else if (code == op["put"]) { print acc }
     else if (code == op["st"]) { mem[addr] = acc }
     else if (code == op["ld"]) { acc = mem[addr] }
     else if (code == op["add"]) { acc += mem[addr] }
     else if (code == op["sub"]) { acc -= mem[addr] }
     else if (code == op["jpos"]) { if (acc > 0) pc = addr }
     else if (code == op["jz"]) { if (acc == 0) pc = addr }
      else if (code == op["j"])
                                  { pc = addr }
      else if (code == op["halt"]) { pc = -1 }
     else
                                   \{ pc = -1 \} # halt if invalid
   }
}
```

The associative array symtab records memory locations for labels. If there is no label for an input line, symtab[""] is set.

Labels start in column one; operators are preceded by white space. Pass 1 sets the field separator variable FS to the regular expression $[\t]+$. This causes every maximal sequence of spaces and tabs in the current input line to be a field separator. In particular, leading white space is now treated as a field separator, so \$1 is always the label and \$2 is

always the operator.

Because the "op code" for const is zero, the single assignment

mem[nextmem++] = 1000 * op[\$1] + \$2 # pack into word

can be used to store both constants and instructions in pass 2.

Exercise 7-1. Modify asm to print the listing of memory and program shown above. \Box

Exercise 7-2. Augment the interpreter to print a trace of the instructions as they are executed. \Box

Exercise 7-3. To get an idea of scale, add code to handle errors, deal with a richer set of conditional jumps, etc. How would you handle literal operands like add =1 instead of forcing the user to create a cell called one? \Box

Exercise 7-4. Write a disassembler that converts a raw memory dump into assembly language. \Box

7.2 A Language for Drawing Graphs

The lexical and syntactic simplicity of our assembly language made its analysis easy to do with field splitting. This same simplicity also appears in some higher-level languages. Our next example is a processor for a prototype language called graph, for plotting graphs of data. The input is a graph specification in which each line is a data point or labeling information for the coordinate axes. Data points are x-y pairs, or y values for which a default sequence of x values 1, 2, 3, etc., is to be generated. Labeling information consists of a keyword and parameter values like

label caption xlabel caption ylabel caption

Such lines can appear in any order, so long as they precede the data. They are all optional.

The processor reads the data and produces a Python program together with a temporary file containing the data in the right format. Running the Python program produces a nicely formatted graph. This is a reasonable division of labor: Awk is well suited for simple processing, while Python plotting libraries like Matplotlib do an excellent job of displaying information. For example, this input:

```
title US Traffic Deaths by Year
xlabel Year
ylabel Traffic deaths
1900 36
1901 54
1902 79
1903 117
1904 172
...
2017 37473
2018 36835
2019 36355
2020 38824
2021 42915
```

produces the output shown in Figure 7-1.

The graph processor operates in two phases. The BEGIN block generates boilerplate Python commands; then each line of the data file is read and converted into the right format,



Figure 7-1: Python-generated graph

using patterns to recognize the different types of statements. Finally, the END block produces the show statement that causes the graph to be plotted.

```
# graph - generate Python program to draw a graph
awk '
BEGIN {
  print "import matplotlib.pyplot as plt"
  print "import pandas as pd"
  print "df = pd.read_csv(\"temp\", sep=\" \")"
  print "plt.scatter(df[\"col1\"],df[\"col2\"])"
  print "col1 col2" >"temp"
}
/xlabel|ylabel|title/ {
  label = $1; $1 = ""
  printf("plt.%s(\"%s\")\n", label, $0)
  next
}
NF == 1 { print ++n, $1 > "temp" }
NF == 2 { print $1, $2 >"temp" }
END { print "plt.show()" }
' $*
```

The graph language falls naturally into the pattern-directed model of computation that Awk itself supports: the specification statements are keywords with values. This style is a good start for any language design; it seems easy for people to use, and it is certainly easy to process. Our language graph is an extremely simplified version of the graph-plotting language grap, by Jon Bentley and Brian Kernighan, which is a preprocessor for the pic picturedrawing language. The same data and a description almost identical to that above produces the graph of Figure 7-2 when run through grap, pic, and troff:



Traffic Deaths by Year

Figure 7-2: Grap-generated graph

Awk is good for designing and experimenting with little languages. If a design proves suitable, a production version can be recoded in a more efficient language like C or Python. In some cases, the prototype version itself may be suitable for production use. These situations typically involve sugar-coating or specializing an existing tool.

A specific instance is the preparation of specialized graphs like scatter-plot matrices, dotcharts (a form of histogram), boxplots, and pie-charts. In the past, we have used Awk programs to translate simple languages into grap commands; today we would more likely generate Python, as we did above.

7.3 A Sort Generator

The Unix sort command is versatile *if* you know how to use it, but it's hard to remember all the options. So as another exercise in little-language design, we will develop a language sortgen to generate sort commands from a more English-like specification. The sortgen processor generates a sort command but does not run it — that task is left to the user, who may want to review the command before invoking it.

The input to sortgen is a little language: a sequence of words and phrases describing sort options like the field separator, the sort keys, and the nature and direction of comparisons. The goal is to cover the common cases with a forgiving syntax. For example, given this input:

```
descending numeric order
```

the output is

sort -rn

As a more complicated example, with this description:

```
field separator is ,
primary key is field 1
    increasing alphabetic
secondary key is field 5
    reverse numeric
```

sortgen produces a sort command equivalent to the first one in Chapter 5:

sort -t',' -k1 -k5rn

The heart of sortgen is a set of rules to translate words and phrases describing sort options into corresponding flags for the sort command. The rules are implemented by pattern-action statements in which the patterns are regular expressions that match the phrases describing sort options; the actions compute the appropriate flags for the sort command. For instance, any mention of "unique" or "discard identical" is taken as a request for the -u option, which discards duplicate items. Similarly, the field separator character is assumed to be either a tab or a single character that appears somewhere on a line containing some form of the word "separate."

The hardest part is processing multiple sort keys. Here the magic word is "key," which has to appear in the input. When it does, the next number is the sort key. (We are ignoring the possibility of a second number that identifies the end of this key.) Each mention of "key" starts collection of options for a new key. Per-key options include ignoring spaces (-b), dictionary order (-d), folding upper and lower case together (-f), numeric order (-n), and reversal (-r).

```
# sortgen - generate a sort command
#
    input: sequence of lines describing sorting options
    output: Unix sort command with appropriate arguments
#
BEGIN { key = 0 }
/no |not |n't / {
   print "error: can't do negatives:", $0 >"/dev/stderr"
   ok = 1
}
# rules for global options
\{ ok = 0 \}
/uniq|discard.*(iden|dupl)/ { uniq = " -u"; ok = 1 }
/key/
       { key++; dokey(); ok = 1 } # new key; must come in order
/separ.*tab|tab.*sep/
                        \{ sep = "t' \setminus t'"; ok = 1 \}
/separ/ { for (i = 1; i <= NF; i++)</pre>
             if (length(\$i) == 1)
                sep = "t'" $i "'"
          ok = 1
        }
```

```
# rules for each key
/dict/
                                 { dict[key] = "d"; ok = 1 }
/ignore.*(space|blank)/
                                  { blank[key] = "b"; ok = 1 }
                                  { fold[key] = "f"; ok = 1 }
/fold|case/
/num/
                                  { num[key] = "n"; ok = 1 }
/rev|descend|decreas|down|oppos/ { rev[key] = "r"; ok = 1 }
/forward|ascend|increas|up|alpha/ { next } # sort's default
!ok { printf("error: can't understand: %s\n", $0) >"/dev/stderr" }
END {
                             # print flags for each key
  cmd = "sort" uniq
  flag = dict[0] blank[0] fold[0] rev[0] num[0] sep
  if (flag) cmd = cmd " -" flag
   for (i = 1; i <= key; i++)</pre>
     if (pos[i] != "") {
         flag = pos[i] dict[i] blank[i] fold[i] rev[i] num[i]
         if (flag) cmd = cmd " -k" flag
      }
  print cmd
}
function dokey( i) { # determine position of key
   for (i = 1; i <= NF; i++)
      if ($i ~ /^[0-9]+$/) {
         pos[key] = $i  # sort keys are 1-origin
        break
      }
  if (pos[key] == "")
     printf("error: invalid key spec: %s\n", $0) > "/dev/stderr"
}
```

To avoid dealing with input like "don't discard duplicates" or "no numeric data," the first pattern of sortgen rejects lines that appear to be phrased negatively. Subsequent rules deal with the global options, then with those that apply only to the current key. The program informs the user of any line it was unable to understand.

This program is still easy to fool, of course, but if one is trying to get the right answer, not to provoke an error, sortgen is already useful.

Exercise 7-5. Write a version of sortgen that provides access to all the facilities of the sort command on your system. Detect inconsistent requests, such as sorting numerically and in dictionary order simultaneously. \Box

Exercise 7-6. How much more accurate can you make sortgen without making its input language significantly more formal?

Exercise 7-7. Write a program that performs the inverse function of translating a sort command into an English sentence. Run sortgen on its output. \Box

7.4 A Reverse-Polish Calculator

We're now going to write several simple calculator programs to illustrate a variety of approaches and Awk techniques.

Suppose we want a calculator program for balancing a checkbook or evaluating arithmetic expressions. Awk itself is perfectly reasonable for such calculations except that we have to re-run it each time the program changes. We need a program that will read and evaluate expressions as they are typed.

To avoid writing a parser, we could require the user to write expressions in reverse-Polish notation. (It's called "reverse" because operators follow their operands, and "Polish" after the Polish logician Jan Łukasiewicz, who first proposed the notation around 1924.) The normal "infix" expression

(1 + 2) * (3 - 4) / 5

is written in reverse Polish as

1 2 + 3 4 - * 5 /

No parentheses are needed — expressions are unambiguous if the number of operands taken by each operator is known. Reverse-Polish expressions are easy to parse and evaluate using a stack and, as a consequence, programming languages like Forth and Postscript, and some early pocket calculators, use this notation.

Our first calculator, calc1, evaluates arithmetic expressions written in reverse-Polish notation, with all operators and operands separated by spaces.

```
# calc1 - reverse-Polish calculator, version 1
#
   input: arithmetic expressions in reverse Polish
#
  output: values of expressions
 for (i = 1; i <= NF; i++) {
{
        if ($i ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/) {
            stack[++top] = $i
        } else if ($i == "+" && top > 1) {
            stack[top-1] += stack[top]; top--
        } else if ($i == "-" && top > 1) {
            stack[top-1] -= stack[top]; top--
        } else if ($i == "*" && top > 1) {
            stack[top-1] *= stack[top]; top--
        } else if ($i == "/" && top > 1) {
            stack[top-1] /= stack[top]; top--
        } else if ($i == "^" && top > 1) {
            stack[top-1] ^= stack[top]; top--
        } else {
            printf("error: cannot evaluate %s\n", $i)
            top = 0
            next
        }
    }
    if (top == 1) {
        printf("\t%.8g\n", stack[top--])
    } else if (top > 1) {
       printf("error: too many operands\n")
        top = 0
    }
}
```

If a field is a number, it is pushed onto a stack; if it is an operator, the proper operation is done to the operands on the top of the stack. The value at the top of the stack is printed and popped at the end of each input line.

For the input

1 2 + 3 4 - * 5 /

calc1 gives the answer -0.6.

Our second reverse-Polish calculator provides user-defined variables and access to a handful of arithmetic functions. Variable names consist of a letter followed by letters or digits; the special syntax var= pops the value on the top of the stack and assigns it to the variable var. If the input line ends with an assignment, no value is printed. Thus a typical interaction might look like this (program output is indented):

```
0 -1 atan2 pi=
pi
3.1415927
355 113 / x= x
3.1415929
x pi /
1.0000001
2 sqrt
1.4142136
```

The program is a straightforward extension of the previous one; it appears in Figure 7-3 on the next page.

Exercise 7-8. Add built-in variables for standard values like π and e to calc2. Add a built-in variable for the result of the last input line. Add stack-manipulation operators to duplicate the top of the stack and to swap the top two items. \Box

7.5 A Different Approach

Another approach to writing a calculator takes advantage of the fact that Awk already does a fine job of evaluating expressions of all types; it's well defined and documented, so there's no need to learn another language. Rather than writing a parser from scratch, we can pipe commands into an instance of Awk and have it do the computations. The version here was inspired by Jon Bentley's *hawk*, which in turn is a play on the *hoc* calculator in *The Unix Programming Environment* by Kernighan and Pike.

The hawk program reads a line at a time, appends it to the previous lines, puts all the lines into a file, then runs Awk on that file; it's a program that writes a program.

This example session illustrates the use of Awk built-in functions:

```
# calc2 - reverse-Polish calculator, version 2
      input: expressions in reverse Polish
#
      output: value of each expression
{ for (i = 1; i <= NF; i++) {
      if ($i ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/) {
          stack[++top] = $i
      } else if ($i == "+" && top > 1) {
          stack[top-1] += stack[top]; top--
      } else if ($i == "-" && top > 1) {
          stack[top-1] -= stack[top]; top--
      } else if ($i == "*" && top > 1) {
          stack[top-1] *= stack[top]; top--
      } else if ($i == "/" && top > 1) {
          stack[top-1] /= stack[top]; top--
      } else if ($i == "^" && top > 1) {
          stack[top-1] ^= stack[top]; top--
      } else if ($i == "sin" && top > 0) {
          stack[top] = sin(stack[top])
      } else if ($i == "cos" && top > 0) {
          stack[top] = cos(stack[top])
      } else if ($i == "atan2" && top > 1) {
          stack[top-1] = atan2(stack[top-1],stack[top]); top--
      } else if ($i == "log" && top > 0) {
          stack[top] = log(stack[top])
      } else if ($i == "exp" && top > 0) {
          stack[top] = exp(stack[top])
      } else if ($i == "sqrt" && top > 0) {
          stack[top] = sqrt(stack[top])
      } else if ($i == "int" && top > 0) {
          stack[top] = int(stack[top])
      } else if ($i in vars) {
          stack[++top] = vars[$i]
      } else if ($i ~ /^[a-zA-Z][a-zA-Z0-9]*=$/ && top > 0) {
          vars[substr($i, 1, length($i)-1)] = stack[top--]
      } else {
          printf("error: cannot evaluate %s\n", $i)
          top = 0
          next
      }
 }
  if (top == 1 && $NF !~ /\=$/) {
     printf("\t%.8g\n", stack[top--])
  } else if (top > 1) {
     printf("error: too many operands\n")
      top = 0
 }
}
```



Here's the implementation of hawk itself:

```
/./ { # ignore blank lines
    f = "hawk.temp"
    hist[++n] = "prev = " $0
    print "BEGIN {" >f
    for (i = 1; i <= n; i++)
        print hist[i] >f
    if ($0 !~ "=")
        print "print \" \" prev" >f
    print "}" >f
    close(f)
    system("awk -f " f)
}
```

This approach gives interactive access to all the expression-evaluation capabilities of Awk. On the flip side, there are some drawbacks. The most serious is that any error in the input tends to leave a chunk of non-working code in the history that might prevent further computations from working. Remedying that is a good exercise.

The other drawback is that each new computation runs all the previous ones, so in effect evaluation time grows quadratically with the number of expressions evaluated. In theory that's a problem, but in practice not at all; one would not use this scheme for anything that required serious computing time.

As an aside, some languages, notably Python, provide a "read-evaluate-print loop" or "REPL": you can type at them and they will interpret what you type on the fly. Awk doesn't support this mode, but hawk is a step in that direction.

Exercise 7-9. Fix hawk to recover gracefully from errors.

Exercise 7-10. Modify hawk to pipe into Awk rather than using a temporary file and re-computing the expressions. \Box

7.6 A Recursive-Descent Parser for Arithmetic Expressions

So far, all of the languages we have considered in this chapter have had a syntax that was easy to analyze. Most high-level languages, however, have operators at several different precedence levels, nested structures like parentheses and if-then-else statements, and other constructions that require more powerful parsing techniques than field splitting or regular expression pattern matching. It is possible to process such languages in Awk by writing a full-fledged parser, as one would in any language. In this section we will construct a program to evaluate arithmetic expressions in the familiar "infix" notation; this is a useful precursor to the much larger parser in the next section.

The key ingredient in a recursive-descent parser is a set of recursive parsing routines, each of which is responsible for identifying, in the input, strings generated by a nonterminal in the grammar. Each routine calls in turn upon others to help out in the task until the terminal level is reached, at which point actual tokens of input are read and categorized. The recursive, top-down nature of this method of parsing leads to the name "recursive descent."

The structure of the parsing routines closely matches the grammatical structure of the language. It is possible to convert a grammar into a parser mechanically, but to do that, the grammar needs to be in a suitable form. Compiler-generator programs like Yacc will do this for you; for details, see Section 4.4 of *Compilers: Principles, Techniques, and Tools* (second edition, 2007), by Aho, Ullman and Sethi. Arithmetic expressions with the operators +, -, *, and / can be described by a grammar in the same style as the one we used in Section 6.1:

```
\begin{array}{rccc} expr & \rightarrow & term \\ & expr + term \\ expr - term \\ term & \rightarrow & factor \\ term * factor \\ term / factor \\ factor & \rightarrow & number \\ & & (expr ) \end{array}
```

This grammar captures not only the form of arithmetic expressions but also the precedences and associativities of the operators. For example, an *expr* is the sum or difference of *terms*, but a *term* is made up of *factors*, which assures that multiplication and division are dealt with before addition or subtraction.

We can think of the process of parsing as one of diagramming a sentence, the opposite of the generation process discussed in Chapter 6. For example, the expression $1 + 2 \times 3$ is parsed like this:



To make an infix evaluator, we need a parser for expressions. With a little effort, the grammar can be used to construct the parser and organize the program as well. A function is written to process each nonterminal in the grammar: the program uses the function expr to process *terms* separated by plus or minus signs, the function term to process *factors* separated by multiplication or division signs, and the function factor to recognize numbers and process parenthesized *exprs*.

In the following program, each input line is taken as a single expression, which is evaluated and printed. For infix notation, it's a nuisance to have to put spaces around operators and parentheses, so this version uses gsub to insert the spaces for us. The input line is edited, then split into an array op. The variable f points to the next field of op to be examined, which is the next operator or operand.

```
# calc3 - infix calculator
#
   input: expressions in standard infix notation
     output: value of each expression
#
NF > 0 \{
   gsub(/[+\-*\/()]/, " & ") # insert spaces around operators
                       # and parentheses
   nf = split(\$0, op)
   f = 1
   e = expr()
   if (f \le nf)
      printf("error at %s\n", op[f])
   else
       printf("\t%.8g\n", e)
}
function expr( e) {
                         # term | term [+-] term
   e = term()
   while (op[f] == "+" || op[f] == "-")
       e = op[f++] == "+"? e + term(): e - term()
   return e
}
function term( e) {
                     # factor | factor [*/] factor
   e = factor()
   while (op[f] == "*" || op[f] == "/")
       e = op[f++] == "*" ? e * factor() : e / factor()
   return e
}
if (op[f] ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/) {
       return op[f++]
   } else if (op[f] == "(") {
       f++
       e = expr()
       if (op[f++] != ")")
           printf("error: missing ) at %s\n", op[f])
       return e
    } else {
       printf("error: expected number or ( at %s\n", op[f])
       return 0
   }
}
```

The regular expression / [+ - + / ()] / is a bit messier than normal because we have to quote two characters: the minus sign because it normally indicates a range of characters, and the slash because the Awk parser would treat an unquoted slash as the end of the regular expression.

Exercise 7-11. Construct a set of inputs to test calc3 thoroughly.

Exercise 7-12. Add exponentiation, built-in functions, and variables to the infix calculator calc3. How does the implementation compare to the reverse-Polish version? \Box

Exercise 7-13. Improve the error-handling performance of calc3.

7.7 A Recursive-Descent Parser for a Subset of Awk

In this section, we develop a recursive-descent translator for a small subset of Awk, written in Awk itself. The part that deals with arithmetic expressions is essentially the same as in the previous section. To make the exercise more realistic, we have chosen to generate C code as the target program, with function calls replacing Awk's operators. This is partly to illustrate the principles of syntax-directed translation, and partly to suggest a way to create a "C version" of Awk that could run faster and be easier to extend.

The general approach is to replace every arithmetic operator by a function call; for example, x=y becomes assign(x, y), and x+y becomes eval("+", x, y). The main input loop is expressed as a while that calls a function getrec to read each input line and split it into fields. Thus,

is translated into this C code:

```
assign(x, num((double)0));
assign(y, num((double)1));
while (getrec()) {
    if (eval(">", field(num((double)1)), x)) {
        if (eval("==", x, eval("+", y, num((double)1)))) {
            assign(x, num((double)1));
            assign(y, eval("*", x, num((double)2)));
        } else {
            print(x, array(z, x));
        }
    }
    if (eval(">", NR, num((double)1))) {
        print(field(num((double)1)));
    }
}
print (NR);
```

We can begin designing the front end of a processor by writing a grammar for the input language. Using the notation of Section 6.1, our subset of Awk has the grammar shown in Figure 7-4. The notation "" stands for the null string and | separates alternatives.

For example, the function for program looks for an optional BEGIN action, followed by a list of pattern-action statements, followed by an optional END action.

In our recursive-descent parser, lexical analysis is done by a routine called advance, which finds the next token and assigns it to the variable tok. Output is produced each time a

program	\rightarrow	opt-begin pa-stats opt-end
opt-begin	\rightarrow	BEGIN statlist ""
opt-end	\rightarrow	END statlist ""
pa-stats	\rightarrow	statlist pattern pattern statlist
pattern	\rightarrow	expr
statlist	\rightarrow	{ stats }
stats	\rightarrow	stat stats ""
stat	\rightarrow	<pre>print exprlist if (expr) stat opt-else while (expr) stat statlist ident = expr</pre>
opt-else	\rightarrow	else stat " "
exprlist	\rightarrow	expr expr , exprlist
expr	\rightarrow	number ident \$expr (expr) expr < expr expr <= expr expr > expr expr + expr expr - expr expr * expr expr / expr expr % expr
ident	\rightarrow	name name [expr] name (exprlist)

Figure 7-4: Grammar for a subset of Awk

stat is identified; lower-level routines return strings that are combined into larger units. An attempt has been made to keep the output readable by inserting tabs; the proper level of nesting is maintained in the variable nt. The parser prints error messages to /dev/stderr, which is an output stream separate from the standard output /dev/stdout.

At about 170 lines, the program is rather long; it is displayed on the three following pages. It is by no means complete — it does not parse all of Awk, nor does it generate all of the C code that would be needed even for this subset — and it is not at all robust. But it does demonstrate how the whole thing might be done, and it also shows the structure of a recursive-descent translator for a nontrivial fraction of a real language.

There have been real-life examples of translating Awk into other languages. Chris Ramming's awkcc is still available; it translates into C, rather in the style of function calls illustrated above. Brian Kernighan wrote a translator into C++, which provided some significant notational conveniences — array subscripts were overloaded to allow strings as subscripts, as in Awk, and Awk variables were first-class citizens of the program — but it was more a proof of concept than anything useful.

Exercise 7-14. Modify the Awk parser to generate some other language than C; for example, Python would be an interesting target. \Box

```
# awk.parser - recursive-descent translator for awk subset
# input: awk program (very restricted subset)
 output: C code to implement the awk program
#
BEGIN { program() }
function advance() {
                      # lexical analyzer; returns next token
   if (tok == "(eof)") return "(eof)"
   while (length(line) == 0)
       if (getline line == 0)
          return tok = "(eof)"
   match(line, /^-?([0-9]+\.?[0-9]*|\.[0-9]+)/) || # number
      match(line, /^(<|<=|==|!=|>=|>)/) ||
                                              # relational
      match(line, /^./)) {
                                          # everything else
          tok = substr(line, 1, RLENGTH)
          line = substr(line, RLENGTH+1)
          return tok
   }
   error("line " NR " incomprehensible at " line)
}
l
                  # read next token if s == tok
function eat(s) {
   if (tok != s) error("line " NR ": saw " tok ", expected " s)
   advance()
function nl() {
                   # absorb newlines and semicolons
   while (tok == "\n" || tok == ";")
       advance()
}
function error(s) { print "Error: " s > "/dev/stderr"; exit 1 }
function program() {
   advance()
   if (tok == "BEGIN") { eat("BEGIN"); statlist() }
   pastats()
   if (tok == "END") { eat("END"); statlist() }
   if (tok != "(eof)") error("program continues after END")
}
function pastats() {
   gen("while (getrec()) {"); nt++
   while (tok != "END" && tok != "(eof)") pastat()
   nt--; gen("}")
if (tok == "{")
                      # action only
      statlist()
   else {
                      # pattern-action
       gen("if (" pattern() ") {"); nt++
       if (tok == "{") statlist()
       else
                       # default action is print $0
         gen("print(field(0));")
      nt--; gen("}")
   }
}
function pattern() { return expr() }
function statlist() {
   eat("{"); nl(); while (tok != "}") stat(); eat("}"); nl()
}
```

```
function stat() {
   if (tok == "print") { eat("print"); gen("print(" exprlist() ");") }
   else if (tok == "if") ifstat()
   else if (tok == "while") whilestat()
   else if (tok == "{") statlist()
   else gen(simplestat() ";")
   nl()
}
function ifstat() {
   eat("if"); eat("("); gen("if (" expr() ") {"); eat(")"); nl(); nt++
   stat()
   if (tok == "else") {  # optional else
       eat("else")
       nl(); nt--; gen("} else {"); nt++
      stat()
   }
   nt--; gen("}")
}
function whilestat() {
   eat("while"); eat("("); gen("while (" expr() ") {"); eat(")"); nl()
   nt++; stat(); nt--; gen("}")
}
function simplestat( lhs) { # ident = expr | name(exprlist)
   lhs = ident()
   if (tok == "=") {
       eat("=")
       return "assign(" lhs ", " expr() ")"
   } else return lhs
}
function exprlist( n, e) { # expr , expr , ...
   e = expr()  # has to be at least one
   for (n = 1; tok == ","; n++) {
       advance()
       e = e ", " expr()
   }
   return e
}
function expr( e) {
                       # rel | rel relop rel
   e = rel()
   while (tok ~ /<|<=|==|!=|>=|>/) {
       op = tok
       advance()
       e = sprintf("eval(\"%s\", %s, %s)", op, e, rel())
   }
   return e
}
function rel( op, e) {
                               # term | term [+-] term
   e = term()
   while (tok == "+" || tok == "-") {
      op = tok
       advance()
      e = sprintf("eval(\"%s\", %s, %s)", op, e, term())
   }
   return e
}
```

```
# fact | fact [*/%] fact
function term( op, e) {
   e = fact()
   while (tok == "*" || tok == "/" || tok == "%") {
       op = tok
       advance()
       e = sprintf("eval(\"%s\", %s, %s)", op, e, fact())
    }
   return e
}
function fact( e) {
                               # (expr) | $fact | ident | number
   if (tok == "(") {
       eat("("); e = expr(); eat(")")
       return "(" e ")"
    } else if (tok == "$") {
       eat("$")
        return "field(" fact() ")"
    } else if (tok ~ /^[A-Za-z_][A-Za-z_0-9]*/) {
       return ident()
    } else if (tok ~ /^-?([0-9]+\.?[0-9]*|\.[0-9]+)/) {
        e = tok
        advance()
       return "num((double)" e ")"
    } else {
       error("unexpected " tok " at line " NR)
    }
}
function ident( id, e) {  # name | name[expr] | name(exprlist)
    if (!match(tok, /^[A-Za-z_][A-Za-z_0-9]*/))
        error("unexpected " tok " at line " NR)
    id = tok
    advance()
   if (tok == "[") {
                              # array
        eat("["); e = expr(); eat("]")
        return "array(" id ", " e ")"
    } else if (tok == "(") { # function call
        eat("(")
        if (tok != ")") {
           e = exprlist()
            eat(")")
        } else eat(")")
       return id "(" e ")" # calls are statements
    } else {
       return id
                              # variable
    }
}
```

7.8 Summary

UNIX provides any number of specialized languages that make it easy to express computations in one focused domain. Regular expressions are the most obvious example, a notation for specifying patterns of text that is used in core tools like grep, sed, and awk; with somewhat different syntax, they also appear in shell wild-card patterns for filename matching.

The shell itself (whichever one you use) is also a specialized language, aimed at making it easy to run programs.

The document preparation tools like troff and preprocessors like eqn and tbl are all languages as well, and the preprocessors were explicitly constructed as languages.

Creating a specialized language can be a productive approach to a programming task. Awk is convenient for translating languages in which lexical analysis and parsing can be done with field splitting and regular expression pattern matching. Associative arrays are good for storing symbol-table information. The pattern-action structure is well suited to patterndirected languages.

The design choices for new languages in new application areas are difficult to make without some experimentation. In Awk it is easy to construct prototypes for feasibility experiments. The results may suggest modifications to an initial design before a large investment in implementation has been made. Once a successful prototype processor has been created, it is relatively straightforward to transcribe the prototype into a production model using compilerconstruction tools like yacc and lex, and programming languages like C or Python. This page intentionally left blank

8

Experiments with Algorithms

Often the best way to understand how something works is to build a small version and do some experiments. This is particularly true for algorithms: writing code illuminates and clarifies issues that are too easily glossed over with pseudo-code. Furthermore, the resulting programs can be tested to ensure that they behave as advertised, which is not true of pseudocode.

Awk can be a good tool for this kind of experimentation. If a program is written in Awk, it's easy to concentrate on the algorithm instead of language details. If the algorithm is ultimately to be part of a larger program, it may be more productive to get a prototype working in isolation first. Small Awk programs are also excellent for building a scaffold for debugging, testing, and performance evaluation, regardless of what language the algorithm itself will ultimately be implemented in.

This chapter illustrates experiments with algorithms. The first half describes three sorting methods that are usually encountered in a first course on algorithms, with Awk programs for testing, performance measurement, and profiling. The second half shows several topological sorting algorithms that culminate in a version of the Unix file-updating utility make.

8.1 Sorting

This section covers three well-known and useful algorithms: insertion sort, quicksort, and heapsort. Insertion sort is short and simple, but efficient only for sorting small numbers of elements; quicksort is one of the best general-purpose sorting techniques; heapsort optimizes worst-case performance. For each of these algorithms, we will give the basic idea, show an implementation, present testing routines, and evaluate the performance. Much of the original code is borrowed from Jon Bentley, as is the inspiration for the scaffolding and profiling programs.

Insertion Sort

Basic idea. Insertion sort is similar to the method of sorting a sequence of cards by picking up the cards one at a time and inserting each card into its proper position in the hand.
Implementation. The following code uses this method to sort an array A[1], ..., A[n] into increasing order. (Technically, we should say "non-decreasing order" since there might be duplicate items, but it sounds a bit pedantic.) The first action reads the input a line at a time into the array; the END action calls isort, then prints the results:

```
# insertion sort
    \{ A[NR] = $0 \}
END { isort(A, NR)
      for (i = 1; i <= NR; i++)
          print A[i]
    }
# isort - sort A[1..n] by insertion
function isort (A, n,
                         i,j,t) {
    for (i = 2; i <= n; i++) {
        for (j = i; j > 1 \&\& A[j-1] > A[j]; j--) {
            # swap A[j-1] and A[j]
            t = A[j-1]; A[j-1] = A[j]; A[j] = t
        }
    }
}
```

Elements 1 through i-1 of A are in their original input order at the beginning of the outer loop of isort. The inner loop moves the element currently in the *i*-th position towards the beginning of the array past any larger elements. At the end of the outer loop, all *n* elements will be in order.

This program will sort numbers or strings equally well. But beware of mixed input — the comparisons will sometimes be surprising because of coercions. For example, 2 comes before 10 numerically, but the string "2" comes after the string "10".

If at the beginning A contains the eight integers

8 1 6 3 5 2 4 7

the array passes through the following configurations:

8	1	6	3	5	2	4	7	
1	8	6	3	5	2	4	7	
1	6	8	3	5	2	4	7	
1	3	6	8	5	2	4	7	
1	3	5	6	8	2	4	7	
1	2	3	5	6	8	4	7	
1	2	3	4	5	6	8	7	
1	2	3	4	5	6	7	8	

The vertical bar separates the sorted part of the array from the elements that have yet to be considered.

Testing. How should we test isort? We could just type at it to see what happens. That's a necessary first step, of course, but for a program of any size it's not a substitute for more careful testing. A second possibility is to generate a large number of sets of random

numbers and check the outputs. That's certainly an improvement, but we can do even better with a small set of tests by a systematic attack on places where code usually goes wrong — the boundaries and special cases. For sorting routines, those might include the following, among others:

a sequence of length 0 (the empty input) a sequence of length 1 (a single number) a sequence of *n* random numbers a sequence of *n* sorted numbers a sequence of *n* numbers sorted in reverse order a sequence of *n* identical numbers

One of the goals of this chapter is to show how Awk can be used to help with testing and evaluation of programs. Let us illustrate by mechanizing test generation and evaluation of results for the sorting routines.

There are two distinct approaches, each with its advantages. The first might be called "batch mode": write a program to execute a pre-planned set of tests, exercising the sort function as suggested above. The following routines generate data and check the results. In addition to isort itself, there are functions for creating arrays of various types of data and for checking whether the array is sorted.

```
# batch test of sorting routines
BEGIN {
    print " 0 elements"
    isort(A, 0); check(A, 0)
    print " 1 element"
    genid(A, 1); isort(A, 1); check(A, 1)
    n = 10
             " n " random integers"
    print "
    genrand(A, n); isort(A, n); check(A, n)
    print " " n " sorted integers"
    gensort(A, n); isort(A, n); check(A, n)
             " n " reverse-sorted integers"
    print "
    genrev(A, n); isort(A, n); check(A, n)
    print " " n " identical integers"
    genid(A, n); isort(A, n); check(A, n)
}
function isort (A, n,
                     i,j,t) {
    for (i = 2; i \le n; i++) {
        for (j = i; j > 1 \&\& A[j-1] > A[j]; j--) {
            # swap A[j-1] and A[j]
            t = A[j-1]; A[j-1] = A[j]; A[j] = t
        }
    }
}
```

```
# test-generation and sorting routines...
function check(A,n,
                    i) {
    for (i = 1; i < n; i++) {
        if (A[i] > A[i+1])
            printf("array is not sorted, element %d\n", i)
    }
}
function genrand(A, n,
                        i) { # put n random integers in A
    for (i = 1; i \le n; i++)
        A[i] = int(n*rand())
}
function gensort(A,n,
                        i) { # put n sorted integers in A
    for (i = 1; i <= n; i++)
        A[i] = i
}
function genrev(A,n,
                      i) { # put n reverse-sorted integers
    for (i = 1; i <= n; i++)
                             # in A
        A[i] = n+1-i
}
function genid(A,n, i) {
                             # put n identical integers in A
    for (i = 1; i \le n; i++)
        A[i] = 1
}
```

The second approach to testing is somewhat less conventional, but particularly suited to Awk. The idea is to build a framework or scaffolding that makes it easy to run tests interactively. This style is a nice complement to batch testing, especially when the algorithm in question is less well understood than sorting. It's also convenient when the task is debugging, since it's easy to create ad hoc tests interactively.

Specifically, we will design what is in effect a tiny language for creating test data and operations. Since the language doesn't have to do much or deal with a big user population, it doesn't have to be complicated. It's also easy to throw the code away and start over again if necessary.

Our language provides for automatic generation of an array of n elements of some type, for explicit specification of the data array, and, looking ahead to the rest of this chapter, for naming the sort to be exercised. We have omitted the sorting and data generation routines, which are the same as in the previous example.

The basic organization of the program is a sequence of regular expressions that scan the input to determine the type of data and type of sorting algorithm to use. If the input doesn't match any of these patterns, an error message suggests how to use it correctly. This is more useful than merely saying that the input was wrong.

```
# interactive test framework for sort routines
/^[0-9]+.*rand/ { n = $1; genrand(A, n); dump(A, n); next }
/^{[0-9]+.*id} { n = $1; genid(A, n); dump(A, n); next }
/^[0-9]+.*sort/ { n = $1; gensort(A, n); dump(A, n); next }
/^[0-9]+.*rev/ { n = $1; genrev(A, n); dump(A, n); next }
/^data/ { # use data directly from this line
   delete A # clear array, start over
    for (i = 2; i \le NF; i++)
       A[i-1] = $i
   n = NF - 1
   next
}
/q.*sort/ { qsort(A, 1, n); check(A, n); dump(A, n); next }
/h.*sort/ { hsort(A, n); check(A, n); dump(A, n); next }
/i.*sort/ { isort(A, n); check(A, n); dump(A, n); next }
/./ { print "data ... | N [rand|id|sort|rev]; [qhi]sort" }
function dump(A, n) { # print A[1]..A[n]
    for (i = 1; i <= n; i++)
       printf(" %s", A[i])
   printf("\n")
}
# test-generation and sorting routines ...
. . .
```

Regular expressions provide a forgiving input syntax; a phrase like "qsort" will result in a quicksort, while "heap" will result in a heapsort. We can also enter data directly as an alternative to automatic generation; this permits us to test the algorithms on text as well as numbers. To illustrate, here is a short dialog with the code above:

```
10 random

6 4 7 6 6 3 0 2 8 0

isort

0 0 2 3 4 6 6 6 7 8

10 reverse

10 9 8 7 6 5 4 3 2 1

qsort

1 2 3 4 5 6 7 8 9 10

data now is the time for all good men

hsort

all for good is men now the time
```

Performance. The number of operations that isort performs depends on n, the number of items to be sorted, and on how sorted they already are. In the worst case, insertion sort is a *quadratic* algorithm; that is, its running time grows as the square of the number of items being sorted. That means that sorting twice as many elements will take about four times as long. If the items happen to be almost in order already, however, each element is sifted down only a few positions on average, there's much less work to do, so the running time grows linearly, that is, proportionally to the number of items.

The graph below shows how isort performs as a function of the number of elements to be sorted on three kinds of inputs: reverse-sorted, random, and equal-element sequences. We are counting comparisons and exchanges of items, which is a fair measure of the amount of work in a sorting procedure. As you can see, the performance of isort is worse for reversesorted sequences than it is for random sequences and both of these are much worse than equal-element sequences. The performance on a sorted sequence (not shown here) is similar to that for an equal-element sequence.



In summary, insertion sort is good for sorting small numbers of items, but its performance degrades badly as the number of items goes up, except when the input is almost sorted.

We generated the data for this graph and the others in this chapter by adding two counters to each sorting function, one for comparisons and one for exchanges. Here is the version of isort with counters:

```
function isort(A,n, i,j,t) { # insertion sort with counters
for (i = 2; i <= n; i++) {
   for (j = i; j > 1 && ++comp &&
        A[j-1] > A[j] && ++exch; j--) {
            # swap A[j-1] and A[j]
            t = A[j-1]; A[j-1] = A[j]; A[j] = t
        }
}
```

The counting is all done in one place, in the test of the inner for loop. Tests joined by && are evaluated left to right until a term is false. The expression ++comp is always true (pre-incrementing is mandatory here), so comp is incremented precisely once per comparison of array elements, just before the comparison. Then exch is incremented if and only if a pair is out of order.

The following program was used to organize the tests and prepare data for plotting; again, it amounts to a tiny language that specifies parameters.

```
# test framework for sort performance evaluation
#
  input: lines with sort name, type of data, sizes...
#
   output: name, type, size, comparisons, exchanges, c+e
 for (i = 3; i <= NF; i++)
{
       test($1, $2, $i)
}
function test(sort, data, n) {
   comp = exch = 0
   if (data ~ /rand/)
       genrand(A, n)
   else if (data ~ /id/)
       genid(A, n)
   else if (data ~ /rev/)
       genrev(A, n)
   else
       print "illegal type of data in", $0
    if (sort ~ /q.*sort/)
        qsort(A, 1, n)
   else if (sort ~ /h.*sort/)
       hsort(A, n)
    else if (sort ~ /i.*sort/)
        isort(A, n)
    else
        print "illegal type of sort in", $0
   print sort, data, n, comp, exch, comp+exch
}
# test-generation and sorting routines ...
```

The input is a sequence of lines like

isort random 0 20 40 60 80 100 isort ident 0 20 40 60 80 100

and the output consists of lines containing the name, type, size, and counts for each size. The output is fed into one of the graph-drawing programs that were described in Section 7.2.

Exercise 8-1. One missing check: is the output a permutation of the input? Add that test. \Box

Exercise 8-2. The function check is not a strong test. What kinds of errors does it fail to detect? How would you implement more careful checking? \Box

Exercise 8-3. Most of our tests are based on sorting integers. How does isort perform on other kinds of input? How would you modify the testing framework to handle more general data? \Box

Exercise 8-4. We have tacitly assumed that each primitive operation takes constant time. That is, accessing an array element, comparing two values, addition, assignment, and so forth, each take a fixed amount of time. Is this a reasonable assumption for Awk programs? Test it by writing programs that process large numbers of items. \Box

Quicksort

Basic idea. One of the most effective general-purpose sorting algorithms is a divide-andconquer technique called quicksort, devised by C. A. R. Hoare in the early 1960s. To sort a sequence of elements, quicksort partitions the sequence into two subsequences and recursively sorts each of them. In the partition step, quicksort selects an element from the sequence as the partition element and divides the remaining elements into two groups: those less than the partition element, and those greater than or equal to it. These two groups are sorted by recursive calls to quicksort. If a sequence contains fewer than two elements, it is already sorted, so quicksort does nothing to it.

Implementation. There are several ways to implement quicksort, depending on how the partition step is done. Our method is simple to understand, though not necessarily the fastest. Since the algorithm is used recursively, we'll describe the partition step as it acts on a subarray A[left], A[left+1], ..., A[right].

First, to choose the partition element, pick a random number r in the range [left,right]. The element p at position r in the array becomes the partition element. Then swap A[left] with A[r]. During the partition step the array holds the element p in A[left], followed by the elements less than p, followed by the elements greater than or equal to p, followed by the as-yet unprocessed elements:



The index last points to the last element found to be less than p and the index i points to the next unprocessed element. Initially, last is equal to left and i is equal to left+1.

In the partition loop, we compare the element A[i] with p. If $A[i] \ge p$, we just increment i. If $A[i] \le p$, we increment last, swap A[last] with A[i] and then increment i. Once we have processed all elements in the array in this manner, we swap A[left] with A[last].

At this point we have completed the partition step and the array looks like this:



Now we apply the same process to the left and the right subarrays.

Suppose we use quicksort to sort an array with the eight elements

8 1 6 3 5 2 4 7

At the first step we might randomly select 4 as the partition element. The partition step would then rearrange the array around this element like this:

2 1 3 4 5 6 8 7

We would then sort each of the subarrays 213 and 5687 recursively. The recursion ceases when a subarray has less than two elements.

The function qsort that implements quicksort is shown below. This program can be tested using the same testing routines that we gave for insertion sort.

```
# quicksort
    \{ A[NR] = $0 \}
END { gsort(A, 1, NR)
     for (i = 1; i \le NR; i++)
         print A[i]
    }
# qsort - sort A[left..right] by quicksort
function qsort(A,left,right, i,last) {
    if (left >= right) # do nothing if array contains
                        # less than two elements
        return
    swap(A, left, left + int((right-left+1)*rand()))
    last = left # A[left] is now partition element
    for (i = left+1; i <= right; i++)</pre>
         if (A[i] < A[left])
            swap(A, ++last, i)
    swap(A, left, last)
    qsort(A, left, last-1)
    qsort(A, last+1, right)
}
function swap(A,i,j, t) {
    t = A[i]; A[i] = A[j]; A[j] = t
}
```

Performance. The number of operations that qsort performs depends on how evenly the partition element divides the array at each step. If the array is always split evenly, then the running time is proportional to $n \log n$. Thus sorting twice as many elements takes only slightly more than twice as long.

In the worst case every partition step might split the array so that one of the two subarrays is empty. This situation would occur if, for example, all elements were equal. In that case, quicksort becomes quadratic. The graph on the next page shows how qsort performs on the three kinds of inputs we used for insertion sort: reverse-sorted, random, and equal-element sequences. As you can see, the number of operations for the equal-element sequences grows significantly faster than for the two other types.

Exercise 8-5. Add counting statements to qsort to count the number of comparisons and exchanges. Does your data look like ours? \Box

Exercise 8-6. Instead of counting operations, time the program. Does the graph look the same? Try some larger examples. Does the graph still look the same? \Box

Heapsort

Basic idea. A *priority queue* is a data structure for storing and retrieving elements. There are two operations: insert a new element into the queue or extract the largest element from the queue. This suggests that a priority queue can be used to sort: first put all the elements into the queue and then remove them one at a time. Since the largest remaining element is removed at each step, the elements will be withdrawn in decreasing order. This technique underlies heapsort, a sorting algorithm devised by J. W. J. Williams and R. W. Floyd in the early 1960s.



Heapsort uses a data structure called a *heap* to maintain the priority queue. We can think of a heap as a binary tree with two properties:

- 1. The tree has a balance property: the leaves appear on at most two different levels and the leaves on the bottom level (furthest from the root) are as far left as possible.
- 2. The tree is partially ordered: the element stored at each node is greater than or equal to the elements at its children.

Here is an example of a heap with ten elements:



There are two important characteristics of a heap. The first is that if there are n nodes, then no path from the root to a leaf is longer than $\log_2 n$. The second is that the largest element is always at the root ("the top of the heap").

We don't need an explicit binary tree if we simulate a heap with an array A in which the elements at the nodes of the binary tree appear in the array in a "breadth-first" order. That is, the element at the root appears in A[1] and its children appear in A[2] and A[3]. In general, if a node is in A[i], then its children are in A[2i] and A[2i+1], or in just A[2i] if there is only one child. Thus, the array A for the elements shown above would contain:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
76	72	34	59	63	17	29	37	33	28

The partially ordered property of the elements in a heap means that A[i] is greater than or equal to its children at A[2i] and A[2i+1], or to its child at A[2i] if there is only one child. If the elements of an array satisfy this condition, we say that the array has the "heap property."

Implementation. There are two phases to heapsort: building a heap and extracting the elements in order. Both phases use a function called heapify (A, i, j) to sift elements down to their proper level, and thus to give the subarray A[i], A[i+1], ..., A[j] the heap property assuming A[i+1], ..., A[j] already has the property. The basic operation of heapify is to compare A[i] with its children. If A[i] has no children or is greater than its children, then heapify merely returns; otherwise, it swaps A[i] with its largest child and repeats the operation at that child.

In the first phase, heapsort transforms the array into a heap by calling heapify (A, i, n) for *i* going from n/2 down to 1.

At the start of the second phase *i* is set to *n*. Then three steps are executed repeatedly. First, A[1], the largest element in the heap, is exchanged with A[*i*], the rightmost element in the heap. Second, the size of the heap is reduced by one by decrementing *i*. These two steps have the effect of removing the largest element from the heap. Note that in doing so the last n-i+1 elements in the array are now in sorted order. Third, heapify (A, 1, *i*-1) is called to restore the heap property to the first *i*-1 elements of A.

These three steps are repeated until only a single element, the smallest, is left in the heap. Since the remaining elements in the array are in increasing order, the entire array is now sorted. During this process, the array looks like this:



The elements in cells 1 through *i* of the array have the heap property; those in cells i + 1 through *n* are the largest n - i elements sorted in increasing order, and those in "sorted" are all greater than or equal to any in "heap." Initially, i = n and there is no sorted part.

Consider the array of elements shown above, which already has the heap property. In the first step of the second phase we exchange elements 76 and 28:

28 72 34 59 63 17 29 37 33 76

In the second step we decrement the heap size to nine. Then in the third step we restore the heap property to the first nine elements by moving 28 to its proper position in the heap by a sequence of swaps:

72 63 34 59 28 17 29 37 33 | 76

We can visualize this process as sifting the element 28 down a path in the binary tree from the root towards a leaf until the element is moved into a node all of whose children are less than or equal to 28:



In the next iteration, the first step exchanges elements 72 and 33:

33 63 34 59 28 17 29 37 | 72 76

The second step decrements *i* to eight and the third propagates 33 to its proper position:

63 59 34 37 28 17 29 33 | 72 76

The next iteration begins by exchanging 63 and 33, and eventually produces the following configuration:

59 37 34 33 28 17 29 63 72 76

This process continues until the array is sorted.

The program below sorts its input into increasing order using this procedure. For reasons that will become apparent when we discuss profiling in the next section, we have enclosed most single-expression statements in braces.

```
# heapsort
    \{ A[NR] = $0 \}
END { hsort(A, NR)
      for (i = 1; i \le NR; i++)
          { print A[i] }
    }
function hsort(A,n, i) {
    for (i = int(n/2); i \ge 1; i--) \# phase 1
         { heapify(A, i, n) }
    for (i = n; i > 1; i--) {
                                       # phase 2
         { swap(A, 1, i) }
         { heapify(A, 1, i-1) }
    }
}
function heapify(A, left, right,
                                 p,c) {
    for (p = left; (c = 2*p) \leq right; p = c) {
        if (c < right \&\& A[c+1] > A[c])
            { c++ }
        if (A[p] < A[c])
            { swap(A, c, p) }
    }
}
function swap(A,i,j,
                       t) {
    t = A[i]; A[i] = A[j]; A[j] = t
}
```

Performance. The total number of operations of hsort is proportional to $n \log n$, even in the worst case, because each of the heapify operations takes $\log n$ time. Below we see the

number of operations from running hsort on the same sequences we used to evaluate insertion sort and quicksort. Note that equal-element performance is better than quicksort.



The next graph compares the performance of the three sorting algorithms of this section on random input data.



Recall that on random data the performance of isort is quadratic while that of hsort and qsort is $n \log n$. The graph clearly shows the importance of good algorithms: as the number of elements increases the difference in performance between the quadratic and the $n \log n$ programs widens dramatically.

Exercise 8-7. The check function always found that the output of isort was sorted. Will this be true of qsort and hsort? Would it be true when the input is just numbers, or just strings that don't look like numbers? \Box

8.2 Profiling

In the previous section, we evaluated the performance of a sorting program by counting the number of times certain operations were executed. Another effective way to evaluate the performance of a program is to profile it, that is, count the number of times each statement is executed. Many programming environments provide a tool, called a *profiler*, that will print a program with an execution count attached to each statement or function.

We don't have a profiler for Awk, but in this section, we will show how to approximate one with two short programs. The first program, makeprof, makes a profiling version of an Awk program by inserting counting and printing statements into the program. When the profiling program is run on some input, it counts the number of times each statement is executed and creates a file prof.cnts containing these counts. The second program, printprof, attaches the statement counts from prof.cnts to the original program.

To simplify the problem, we will only count the number of times each left brace is "executed" during the run of a program. Often this is good enough because every action and every compound statement is enclosed in braces. Any statement can be enclosed in braces, however, so we can obtain as precise an execution count as we wish by bracketing statements.

Here is the program makeprof that transforms an ordinary Awk program into a profiling program. It inserts a counting statement of the form

_LBcnt[*i*]++;

after the first left brace appearing on the *i*-th input line, and it adds a new END action that prints the values of these counters into prof.cnts, one count per line.

```
# makeprof - prepare profiling version of an awk program
# usage: awk -f makeprof awkprog >awkprog.p
# running awk -f awkprog.p data creates a
# file prof.cnts of statement counts for awkprog
    { sub(/{/, "{ _LBcnt[" ++_numLB "]++; ")
        print
    }
END { printf("END { for (i = 1; i <= %d; i++)\n", _numLB)
        printf("\t\t print _LBcnt[i] > \"prof.cnts\"\n}\n")
    }
```

After running the profiling version of a program on some input data, we can attach the statement counts in prof.cnts to the original program with printprof:

As an example, consider profiling the heapsort program from the end of Section 8.1. To create the profiling version of this program, type the command line

awk -f makeprof heapsort >heapsort.p

The resulting program heapsort.plooks like this:

```
# heapsort
    { _LBcnt[3]++; A[NR] = $0 }
END { _LBcnt[5]++; hsort(A, NR)
     for (i = 1; i <= NR; i++)
          { _LBcnt[7]++; print A[i] }
    }
function hsort(A,n, i) { _LBcnt[10]++;
    for (i = int(n/2); i \ge 1; i--) \# phase 1
         { _LBcnt[12]++; heapify(A, i, n) }
    for (i = n; i > 1; i--) { _LBcnt[13]++;
                                                    # phase 2
        { _LBcnt[14]++; swap(A, 1, i) }
         { _LBcnt[15]++; heapify(A, 1, i-1) }
    }
}
function heapify(A, left, right, p, c) { _LBcnt[18]++;
    for (p = left; (c = 2*p) <= right; p = c) { _LBcnt[19]++;
        if (c < right \&\& A[c+1] > A[c])
            { _LBcnt[21]++; c++ }
        if (A[p] < A[c])
            { _LBcnt[23]++; swap(A, c, p) }
    }
}
function swap(A,i,j, t) { _LBcnt[26]++;
   t = A[i]; A[i] = A[j]; A[j] = t
}
END { for (i = 1; i \le 28; i++)
        print _LBcnt[i] > "prof.cnts"
}
```

As you can see, 13 counting statements have been inserted into the original program, along with a second END section that writes the counts into prof.cnts. Multiple BEGIN and END actions are treated as if they were combined into one in the order in which they appear.

Now, suppose we run heapsort.p on 100 random integers. We can create a listing of the original program with the statement counts resulting from this run by typing the command line

awk -f printprof heapsort

The result is:

```
# heapsort
100
             \{ A[NR] = $0 \}
 1
        END { hsort(A, NR)
               for (i = 1; i \le NR; i++)
100
                   { print A[i] }
             3
        function hsort (A, n,
 1
                               i) {
             for (i = int(n/2); i \ge 1; i--) \# phase 1
50
                  { heapify(A, i, n) }
 99
            for (i = n; i > 1; i--) \{ \# phase 2 \}
99
                  { swap(A, 1, i) }
99
                  { heapify(A, 1, i-1) }
             }
        }
149
        function heapify(A,left,right,
                                            p,c) {
521
             for (p = left; (c = 2*p) \leq right; p = c) {
                 if (c < right \&\& A[c+1] > A[c])
232
                     { c++ }
                 if (A[p] < A[c])
485
                     { swap(A, c, p) }
             }
584
        function swap(A,i,j,
                                t) {
            t = A[i]; A[i] = A[j]; A[j] = t
        }
```

Simplicity, the greatest strength of this implementation, is also its greatest weakness. The program makeprof blindly inserts a counting statement after the first left brace it sees on each line; a more careful makeprof would not put counting statements inside string constants, regular expressions, or comments. It would also be nice to report execution times as well as counts, but that's not feasible with this approach.

Exercise 8-8. Modify the profiler so that counting statements will not be inserted into string constants, regular expressions, or comments. Will your version permit you to profile the profiler? \Box

Exercise 8-9. The profiler doesn't work if there is an exit statement in the END action. Why? Fix it.

8.3 Topological Sorting

In a construction project, some jobs must be done before others can begin. We want to list them so that each job precedes those that must be done after it. In a program library, a program a may call program h. Program h in turn may call programs d and e, and so on. We would like to order the programs so that a program appears before all the programs it calls. These problems and others like them are instances of the problem of *topological sorting*: finding an ordering that satisfies a set of constraints of the form "x must come before y." In a topological sort any linear ordering that satisfies the partial order represented by the constraints is sufficient.

The constraints can be represented by a graph in which the nodes are labeled by the names, and there is an edge from node x to node y if x must come before y. The following graph is an example:



If a graph contains an edge from x to y, then x is called a *predecessor* of y, and y is a *successor* of x. Suppose the constraints come in the form of predecessor-successor pairs where each input line contains x and y representing an edge from node x to node y, as in this description of the graph above:

h а b q f С С h d i е d f b f g h d h е i b

If there is an edge from x to y, then x must appear before y in the output. Given the input above, one possible output is the list

acfhedibg

There are many other linear orders that contain the partial order depicted in the graph; another is

```
cahedifbg
```

The problem of topological sorting is to sort the nodes of a graph so that all predecessors appear before their successors. Such an ordering is possible if and only if the graph does not contain a *cycle*, which is a sequence of edges that leads from a node back to itself. If the input graph contains a cycle, then a topological sorting program should say so and indicate that no linear ordering exists.

Breadth-First Topological Sort

There are many algorithms that can be used to sort a graph topologically. Perhaps the simplest is one that at each iteration removes from the graph a node with no predecessors. If all nodes can be removed from the graph this way, the sequence in which the nodes are removed is a topological sort of the graph. In the graph above, we could begin by removing node a and the edge that comes from it. Then we could remove node c, then nodes f and h in either order, and so on.

Our implementation uses a first-in, first-out data structure called a *queue* to sequence the processing of nodes with no predecessors in a "breadth-first" manner. After all the data has been read in, a loop counts the nodes and places all nodes with no predecessors on the queue. A second loop removes the node at the front of the queue, prints the node name, and decrements the predecessor count of each of its successors. If the predecessor count of any of its

successors becomes zero, those successors are put on the back of the queue. When the front catches up to the back and all nodes have been considered, the job is done. But if some nodes are never put on the queue, those nodes are involved in cycles and no topological sort is possible. When no cycles are present, the sequence of nodes printed is a topological sort.

The first three statements of tsort read the predecessor-successor pairs from the input and construct a successor-list data structure like this:

node	pcnt	scnt	slist
a	- 0	1	h
b	2	1	a
с С	0	2	f.h
d	2	1	i,
e	1	1	d
f	1	2	b, q
q	2	0	
h	2	2	d, e
i	1	1	b

The arrays point and soint keep track of the number of predecessors and successors for each node; slist[x, i] gives the node that is the *i*-th successor of node x. The first line creates an element of point if it is not already present.

```
# tsort - topological sort of a graph
#
    input: predecessor-successor pairs
#
    output: linear order, predecessors first
    { if (!($1 in pcnt))
          pcnt[\$1] = 0
                                  # put $1 in pcnt
      pcnt[$2]++
                                  # count predecessors of $2
      slist[$1, ++scnt[$1]] = $2 # add $2 to successors of $1
    }
END { for (node in pcnt) {
          nodecnt++
          if (pcnt[node] == 0) # if it has no predecessors
              q[++back] = node # queue node
      }
      for (front = 1; front <= back; front++) {</pre>
          printf(" %s", node = q[front])
          for (i = 1; i <= scnt[node]; i++) {</pre>
              if (--pcnt[slist[node, i]] == 0)
                  # queue s if it has no more predecessors
                  q[++back] = slist[node, i]
          }
      if (back != nodecnt)
          print "\nerror: input contains a cycle"
      printf("\n")
    }
```

The implementation of a queue is especially easy in Awk: it's just an array with two subscripts, one for the front and one for the back.

Exercise 8-10. Fix tsort so it can identify and report isolated nodes in the graph.

Depth-First Search

We will construct one more topological sort program in order to illustrate an important technique called depth-first search, which can also be used to solve many other graph problems, including one that arises in the Unix utility make, which we will see in the next section. Depth-first search is another method of visiting the nodes of a graph, even one with cycles, in a systematic manner. In its purest form, it is just a recursive procedure:

dfs(*node*): mark *node* visited for all unvisited successors *s* of *node* do dfs(*s*)

The reason the technique is called depth-first search is that it starts at a node, then visits an unvisited successor of that node, then an unvisited successor of that successor, and so on, plunging as deeply into the graph as quickly as it can. Once there are no unvisited successors of a node, the search retreats to the predecessor of that node and visits another of its unvisited successors in a depth-first search.

Consider the following graph. If a depth-first search starts at node 1, it will visit nodes 1, 2, 3, and 4. At that point, if it starts with another unvisited node such as 5, it will then visit nodes 5, 6, and 7. If it starts at a different place, however, a different sequence of visits will be made.



Depth-first search is useful for finding cycles. An edge like (3,1) that goes from a node to a previously visited ancestor is called a *back edge*. Since a back edge identifies a cycle, to find cycles all we need to do is find back edges. The following function will test whether a graph, stored as a successor-list data structure like that in tsort, contains a cycle reachable from node:

```
# dfs - depth-first search for cycles
function dfs(node, i, s) {
    visited[node] = 1
    for (i = 1; i <= scnt[node]; i++)
        if (visited[s = slist[node, i]] == 0)
            dfs(s)
        else if (visited[s] == 1)
            printf("cycle with back edge (%s, %s)\n", node, s)
        visited[node] = 2
}</pre>
```

This function uses an array visited to determine whether a node has been traversed. Initially, visited[x] is 0 for all nodes. Entering a node x for the first time, dfs sets visited[x] to 1, and leaving x for the last time it sets visited[x] to 2. During the traversal, dfs uses visited to determine whether a node y is an ancestor of the current node (and hence previously visited), in which case visited[y] is 1, or whether y has been previously visited, in which case visited[y] is 2.

Depth-First Topological Sort

The function dfs can easily be turned into a topological sort procedure. If it prints the name of each node once the search from that node is completed, it will generate a list of nodes that is a reverse topological sort, provided again there are no cycles in the graph. The program rtsort prints the reverse of a topological sort of a graph, given a sequence of predecessor-successor pairs as input. It applies depth-first search to every node with no predecessors. The data structure is the same as that in tsort. The code appears in Figure 8-1.

Applied to the predecessor-successor pairs at the beginning of this section, rtsort prints

gbidehafc

Notice that this algorithm detects some cycles explicitly by finding a back edge, while it detects other cycles only implicitly, by failing to print all the nodes, as in this graph:



Exercise 8-11. Modify rtsort to print its output in the usual order, predecessors first. Can you achieve the same effect without modifying rtsort? \Box

8.4 Make: A File Updating Program

A large program may consist of declarations and subprograms that are stored in many separate files, with an involved sequence of processing steps to create a running version. A complex document (like this chapter) may consist of programs, inputs, outputs, graphs and diagrams stored in multiple files, programs to be run and tested, and then interdependent operations to make a printed copy. An automatic updating facility is an invaluable tool for processing such systems of files with a minimum of human and computer time. This section develops a rudimentary updating program, patterned after the Unix make command, that is based on the depth-first search technique of the previous section.

```
# rtsort - reverse topological sort
# input: predecessor-successor pairs
# output: linear order, successors first
    { if (!($1 in pcnt))
         pcnt[$1] = 0
                                 # put $1 in pcnt
     pcnt[$2]++
                                 # count predecessors of $2
     slist[$1, ++scnt[$1]] = $2 # add $2 to successors of $1
    }
END { for (node in pcnt) {
          nodecnt++
          if (pcnt[node] == 0)
             rtsort(node)
      }
     if (pncnt != nodecnt)
         print "error: input contains a cycle"
     printf("\n")
    }
function rtsort (node,
                      i, s) {
   visited[node] = 1
    for (i = 1; i <= scnt[node]; i++) {</pre>
        if (visited[s = slist[node, i]] == 0)
           rtsort(s)
        else if (visited[s] == 1)
            printf("error: nodes %s and %s are in a cycle\n",
                s, node)
    }
   visited[node] = 2
   printf(" %s", node)
   pncnt++ # count nodes printed
}
```

Figure 8-1: Depth-first topological sort

To use the updater, one must explicitly describe what the components of the system are, how they depend upon one another, and what commands are needed to construct them. We'll assume these dependencies and commands are stored in a file, called a makefile, that contains a sequence of rules of the form

name: $t_1 \ t_2 \ \dots \ t_n$ *commands*

The first line of a rule is a dependency relation that states that the program or file *name* depends on the targets $t_1, t_2, ..., t_n$ where each t_i is a filename or another *name*. Following each dependency relation may be one or more lines of *commands* that list the commands necessary to generate *name*. Here is an example of a makefile for a small program with two C files called a.c and b.c, and a yacc grammar file c.y, a typical program-development application.

```
a.o b.o c.o
prog:
        gcc a.o b.o c.o -ly -o prog
a.o:
        prog.h a.c
        gcc -c prog.h a.c
        prog.h b.c
b.o:
        qcc -c proq.h b.c
c.o:
        C.C
        gcc -c c.c
c.c:
        c.y
        yacc c.y
        mv y.tab.c c.c
print:
        pr prog.h a.c b.c c.y
```

The first line states that prog depends on the target files a.o, b.o, and c.o. The second line says that prog is generated by using the C compiler command gcc to link a.o, b.o, c.o, and a yacc library y into the file prog. The next rule (third line) states that a.o depends on the targets prog.h and a.c and is created by compiling these targets; b.o is the same. The file c.o depends on c.c, which in turn depends on c.y, which has to be processed by the yacc parser generator. Finally, the name print does not depend on any target; by convention, for targetless names make will always perform the associated action, in this case printing all the source files with the command pr.

The dependency relations in the makefile can be represented by a graph in which there is an edge from node x to node y whenever there is a dependency rule with x on the left side and y one of the targets on the right. For a rule with no targets, a successorless node with the name on the left is created. For the makefile above, we have the following dependency graph:



We say that x is *older* than y if y was changed after x was last changed. To keep track of ages, we will attach to each x an integer age[x] that represents how long ago x was last modified. The larger the age, the older the file: x is older than y if $age[x] \ge age[y]$.

If we use the dependency relation

n: a b c

we must bring n up to date by first updating a, b, and c, which may in turn require further updates. If any of the targets is neither a name in the makefile nor an existing file, we report the error and quit. Otherwise, we next examine the ages of the targets, and if at least one is newer than n (that is, if n is older than something it depends on), we execute the

commands associated with this dependency relation. After executing the commands, we recompute the ages of all objects. With a dependency relation like

```
print:
pr prog.h a.c b.c c.y
```

that is, one with no targets, we always execute the command associated with this rule and recompute all ages.

The program make takes a *name* as an argument and updates *name* using the following algorithm:

- 1. It finds the rule for *name* in the makefile and recursively updates the targets $t_1, t_2, ..., t_n$ on the right side of the dependency relation for *name*. If for some *i*, t_i is not a name and file t_i does not exist, make aborts the update.
- 2. If, after updating all the t_i 's, the current version of *name* is older than one or more of the t_i 's, or if *name* has no targets, make executes the command lines following the dependency relation for *name*.

In essentially the same manner as in the previous section, make constructs a dependency graph from the dependency relations in the makefile. It uses the Unix command

ls -t

to order the files (newest first) by the time at which each file was last modified. Each filename is entered into the array age and given a time that is its rank in this ordering; the oldest file has the largest rank. If a name is not a file in the current directory, make sets its time to a large value, thus making it old indeed.

Finally, make uses the depth-first search procedure of the last section to traverse the dependency graph. At node n, make traverses the successors of n; if any successor becomes younger than the current age of n, make executes the commands for n and computes a new set of ages. If make discovers that the dependency relation for a name is cyclic, it says so and aborts the update.

To illustrate how make works, suppose we type the command line

\$ make prog

for the first time. Then make will execute the following sequence of commands:

```
gcc -c prog.h a.c
gcc -c prog.h b.c
yacc c.y
mv y.tab.c c.c
gcc -c c.c
gcc a.o b.o c.o -ly -o prog
```

Now if we make a change to b.c and again type

\$ make prog

make will only execute

gcc -c prog.h b.c gcc a.o b.o c.o -ly -o prog

Because the other files have not changed since the last time prog was created, make does not process them. Finally, if we again say

\$ make prog

nothing is executed and the result is

prog is up to date

because nothing has to be done.

```
# make - maintain dependencies
BEGIN {
   while (getline <"makefile" > 0) {
       if ($0 ~ /^[A-Za-z]/) { # $1: $2 $3 ...
           sub(/:/, "")
           if (++names[nm = \$1] > 1)
               error(nm " is multiply defined")
           for (i = 2; i <= NF; i++) # remember targets</pre>
               slist[nm, ++scnt[nm]] = $i
       cmd[nm] = cmd[nm] $0 "\n" # current name
       } else if (NF > 0) {
           error("illegal line in makefile: " $0)
       }
   }
               # compute initial ages
   ages()
   if (ARGV[1] in names) {
       if (update(ARGV[1]) == 0)
           print ARGV[1] " is up to date"
    } else {
       error(ARGV[1] " is not in makefile")
   }
}
function ages(
               f,n,t) {
   for (t = 1; ("ls -t" | getline f) > 0; t++)
       age[f] = t
                        # all existing files get an age
   close("ls -t")
   for (n in names)
       if (!(n in age)) # if n has not been created
           age[n] = 9999 # make n really old
}
```

```
function update(n,
                     changed, i, s) {
    if (!(n in age))
        error(n " does not exist")
    if (!(n in names))
        return 0
    changed = 0
    visited[n] = 1
    for (i = 1; i <= scnt[n]; i++) {</pre>
        if (visited[s = slist[n, i]] == 0)
            update(s)
        else if (visited[s] == 1)
            error(s " and " n " are circularly defined")
        if (age[s] <= age[n])</pre>
            changed++
    }
    visited[n] = 2
    if (changed || scnt[n] == 0) {
        printf("%s", cmd[n])
        system(cmd[n]) # execute cmd associated with n
        ages()
                        # recompute all ages
        age[n] = 0
                     # make n very new
        return 1
    }
    return 0
}
function error(s) { print "error: " s; exit }
```

Exercise 8-12. How many times is the function ages executed on the example?

Exercise 8-13. Add some parameter or macro substitution mechanism so rules can be easily changed.

Exercise 8-14. Add implicit rules for common updating operations; for example, .c files are processed by gcc to make .o files. How can you represent the implicit rules so they can be changed by users? \Box

8.5 Summary

This chapter may have more of the flavor of a basic course in algorithms than instruction in Awk. The algorithms are genuinely useful, however, and we hope that in addition you have seen something of how Awk can be used to support experiments on programs. Our quicksort, heapsort, and topological sort programs were borrowed from Jon Bentley.

Scaffolding is one of the lessons. It often takes no more time to write a small program to generate and control testing or debugging than it does to perform a single test, but the scaffolding can be used over and over to do a much more thorough job. We adapted our approach to scaffolding from Jon Bentley as well. Bentley's books *Programming Pearls* and *More Programming Pearls* are great further reading.

The other aspect is more conventional, though it bears repeating. Awk is good for extracting data from the output of some program and massaging it for another; for example, that is how we converted sorting measurements into grap input and how we folded statement counts into a profile. This page intentionally left blank

Epilogue

9

By now you should be a reasonably adept Awk user, or at least no longer an awkward beginner. As you have studied the examples and written some code of your own, you may have wondered why Awk programs are the way they are, and perhaps wanted to make them better.

The first part of this chapter provides a little history, and discusses the strengths and weaknesses of Awk as a programming language. The second part explores the performance of Awk programs, and suggests some ways of reformulating problems that have become too large for a single program.

9.1 Awk as a Language

We began working on Awk in 1977. At that time the Unix programs that searched text files (grep and sed) only had regular expression patterns, and the only actions were printing selected lines and (with sed) text substitution. There were no fields and no numeric operations. Our goal, as we remember it, was to create a pattern-scanning language that would understand fields, one with patterns to match fields and actions to manipulate them. Initially, we just wanted to do transformations on data, to scan the inputs of programs for validation, and to process the outputs to generate reports or to rearrange them for input to other programs.

The 1977 version had only a few built-in variables and predefined functions. It was designed for writing short programs like those in Chapter 1. Furthermore, it was designed to be used by our immediate colleagues with little instruction, so for regular expressions we used the familiar notation of egrep, which was written by Al Aho, and Michael Lesk's lex, whose code was based on egrep. For the other expressions and statements we used the syntax of C.

Our model was that an invocation would be one or two lines long, typed in and used immediately. Defaults were chosen to match this style. In particular, white space as the default field separator, implicit initializations, and no type declarations for variables were choices that made it possible to write one-liners. We, being the authors, "knew" how the language was supposed to be used, and so we only wrote one-liners. Awk quickly spread to other groups and users pushed hard on the language. We were surprised at how rapidly Awk became popular as a general-purpose programming language; our first reaction to a program that didn't fit on one page was shock and amazement. What had happened was that many people restricted their use of the computer to the shell (the command language) and to Awk. Rather than writing in a "real" programming language, they were stretching the tools they found convenient.

The idea of having each variable maintain both a string and a numeric representation of its value, and use the form appropriate to the context, was an experiment. The goal was to make it possible to write short programs using only one set of operators, but have them work correctly in the face of ambiguity about strings and numbers. The goal was largely met, but there are still occasional surprises for the unwary. The rules in the reference manual for resolving ambiguous cases evolved from user experience.

Associative arrays were inspired by SNOBOL4 tables, although they are not as general. Awk was born on a slow computer with a small memory, and the properties of arrays were a result of that environment. Restricting subscripts to be strings is one manifestation, as is the restriction to a single dimension (even with syntactic sugar to partially simulate multiple dimensions). A more general implementation would allow multidimensional arrays, or permit arrays to be array elements. (Gawk does this.)

Major new facilities were added to Awk in 1985, largely in response to user demand. These additions included dynamic regular expressions, new built-in variables and functions, multiple input streams, and, most importantly, user-defined functions.

The substitution functions, match, and dynamic regular expressions provided useful capabilities with only a small increase in complexity for users.

Before getline, the only kind of input was the implicit input loop implied by the pattern-action statements. That was fairly restrictive. In the original language, a program like the form-letter generator that had more than one source of input required setting a flag variable or some similar trick to read the sources. In the new language, multiple inputs can be naturally read with getlines in the BEGIN section. On the other hand, getline is overloaded, and its syntax doesn't match the other expressions. Part of the problem is that getline needs to return what it reads, and also some indication of success or failure.

The implementation of user-defined functions was a compromise. The chief difficulties arose from the initial design of Awk. We did not have, or want, declarations in the language. One result is the peculiar way of declaring local variables as extra formal parameters. Besides looking strange, this is error prone. In addition, the absence of an explicit concatenation operator, an advantage for short programs, now requires the opening parenthesis of a function call to follow the function name with no intervening spaces. Nevertheless, the new facilities made Awk significantly better for larger applications.

As Awk grew from a pre-teenager to middle age, there were only a few language changes, thanks to a combination of benign neglect, distaste for bloat, and a desire for stability. At the same time, other languages, notably Perl, were becoming popular. Perl provided everything that Awk did and much more, and ran significantly faster; it also had options that let it handle Awk or Awk-like programs easily.

Python, created in 1991, a few years after Perl, has basically taken over the scripting language niche and is one of the most widely used of all languages. It's an easy language to learn, it's expressive and efficient, and it has an enormous set of libraries for almost any programming task you can imagine. Realistically, if you're going to learn only one language, Python is the one. But for small programs typed on the command line, Awk is hard to beat. At the same time, it's good for somewhat larger programs as well. And Gawk, the GNU version, offers a variety of extensions that make it possible to deal conveniently with more programming challenges. For instance, Gawk has source file inclusion, dynamic library linking (so it can be extended by calling C code), and myriad other useful additions.

9.2 Performance

In a way, Awk is seductive — it is easy to write a small program that does what you want, and for modest amounts of data, it will be fast enough, especially when the program itself is still undergoing changes.

But as a working Awk program is applied to bigger and bigger files, it gets slower and slower. Rationally this must be so, but waiting for your results may be too much to bear. There are no simple solutions except to buy faster hardware, but this section contains suggestions that might be helpful.

When programs take too long to run, there are several things to think about doing, besides just putting up with it. First, it is possible that the program can be made faster, either by a better algorithm or by replacing some frequently executed expensive construction with a cheaper one. You have already seen in Chapter 8 how much difference a good algorithm can make — the difference between a linear algorithm and a quadratic one grows dramatically even with modest increases in data. Second, you can use other programs along with Awk, reducing Awk's role. Third, you can rewrite the entire program in some other language that is more suitable.

Before you can improve the behavior of a program, you need to understand where the time is going. Even in languages where each operation is close to the underlying hardware, people's initial estimates of where time is being spent are notoriously unreliable. Such estimates are even trickier in Awk, because many of its operations do not correspond to conventional machine operations. Among these are array indexing, pattern matching, field splitting, string concatenation, and substitution. The instructions that Awk executes to implement these operations vary from computer to computer and implementation to implementation, and so do their relative costs in Awk programs.

Awk has no built-in tools for timing. Thus it's up to the user to understand what's expensive and what's cheap in the local environment. The easiest way to do this is to make differential measurements of various constructs. For example, how much does it cost to read a line or increment a variable? In 1987, we made measurements on a variety of computers, ranging from a PC clone (AT&T 6300) to a "mainframe" of the time (VAX 8550). We ran three programs on an input file of 10,000 lines (500,000 characters) as well as the Unix command wc for comparison. All times are in seconds.

PROGRAM	AT&T 6300+	DEC VAX 11-750	AT&T 3B2/600	SUN-3	DEC VAX 8550
END { print NR }	30	17.4	5.9	4.6	1.6
{n++}; END {print n}	45	24.4	8.4	6.5	2.4
{ i = NF }	59	34.8	12.5	9.3	3.3
wc command	30	8.2	2.9	3.3	1.0

All of those computers are long gone, of course. Modern consumer-grade computers are far faster, and they don't exhibit such a wide range of performance. We have repeated the experiments with Awk and Gawk, and 100 times as much data (1,000,000 lines, 50 megabytes), but on a single computer, a 2015 MacBook Air.

PROGRAM	Awk	Gawk
END { print NR }	2.5	0.13
{n++}; END {print n}	2.6	0.16
{ i = NF }	2.8	0.51
\$1 == "abc"	2.8	0.25
\$1 ~ /abc/	3.1	0.27
wc command	0.27	
grep ^abc	0.80	

This experiment shows vividly how different implementations can have significantly different performance.

A string comparison like \$1 == "abc" costs about the same as the regular expression match $\$1 \sim /abc/$. The cost of matching a regular expression is more or less independent of its complexity, however, while a compound comparison costs more as it gets more complicated. Dynamic regular expressions can be more expensive, since it may be necessary to recreate a recognizer for each test.

Concatenating lots of strings is more expensive:

print \$1 " " \$2 " " \$3 " " \$4 " " \$5

takes 4.4 seconds in Awk, while

print \$1, \$2, \$3, \$4, \$5

takes 3.8 seconds.

As an extreme example, we once advised a user that his attempt to combine all the lines of a million-line file into a single string was doomed. His program was

{ s = s \$0 }

The problem is that Awk's implementation is (intentionally) naive: the new string is created by allocating space for the new string, copying the old string into it, then concatenating the new input to the end of that, which is a quadratic algorithm. Fortunately, his problem was simple to address by dealing with the input a line at a time.

As we hinted earlier, arrays have complex behavior. Accessing an element takes on average a constant amount of time, but in an amortized sense. As the number of elements in an array grows, the internal representation (an array of linked lists) is reorganized to maintain an average constant-time access.

We saw one example of complex array behavior in the charfreq example of Section 3.4, where splitting an input line into individual characters was slower than isolating individual characters with substr.

Another line of attack is to restructure the computation so that some of the work is done by other programs. Throughout this book, we have made extensive use of the system sort command, for example, rather than writing our own sort in Awk. If you have to search a big file to isolate a small amount of data, it might be more efficient to use grep or egrep for the searching and Awk for the processing. If there are a large number of substitutions (for example, the cross-reference program of Section 6.3), you might use a stream editor like sed for that part. In other words, break the job into separate pieces, and apply the most appropriate tool to each piece.

The last resort is to rewrite the offending program in some other language. The guiding principle is to replace the useful built-in features of Awk with subroutines, and otherwise use much the same structure as the original program. Don't attempt to simulate exactly what Awk does. Instead provide just enough for the problem at hand. A useful exercise is to write a small library that provides field-splitting, associative arrays, and regular expression matching; in languages like C that do not provide dynamic strings, you will also want some routines that allocate and free strings conveniently. With this library in hand, converting an Awk program into something that will run faster is quite feasible.

Awk makes easy many things that are hard in conventional languages, by providing features like pattern matching, field splitting, and associative arrays. The penalty paid is that an Awk program using these features, however easy to write, is not as efficient as a carefully written C program for the same task. Frequently efficiency is not critical, and so Awk is both convenient to use, and fast enough.

When Awk isn't fast enough, it is important to measure the pieces of the job, to see where the time is going. The relative costs of various operations differ from machine to machine, but the measurement techniques can be used on any computer. Finally, even though it is less convenient to program in lower-level languages, the same principles of timing and understanding have to be applied, or else the new program will be both harder to write and less efficient.

As one experiment, to be taken in large part as anecdote, we have written the formatter of Section 6.3, reproduced here:

```
# fmt - format text into 60-char lines
/./ { for (i = 1; i <= NF; i++) addword($i) }
/^$/ { printline(); print "" }
END { printline() }
function addword(w) {
    if (length(line) + length(w) > 60)
        printline()
    line = line space w
        space = " "
}
function printline() {
    if (length(line) > 0)
        print line
    line = space = "" # reset for next line
}
```

in nearly 20 other languages, and compared run times along with Gawk, Mawk, and BBAwk. The input was 770,000 lines (110 megabytes) of ASCII text (25 concatenated copies of the King James bible). Figure 9-1 shows the results, sorted in order of total time. For languages like C, C++, and Java that have a separate compilation step, that time is not included. Take these numbers with a big grain of salt, however; most of the programs were written by the

Language	User time	System time	Total time	Source lines
С	1.66	0.13	1.79	31
Mawk	5.51	0.17	5.68	14
C++	5.37	1.60	6.97	34
Gawk	7.97	0.12	8.09	14
Perl	9.88	0.17	10.05	22
Kotlin	6.48	4.02	10.50	43
Java	6.56	4.05	10.61	43
JavaScript	8.53	2.34	10.87	28
Go	7.92	3.90	11.82	36
Python	12.47	0.15	12.62	25
Scala	9.93	3.52	13.45	36
Awk	15.84	0.15	15.99	14
Ruby	21.53	0.23	21.76	21
Lua	23.50	0.17	23.67	27
PHP	22.02	2.18	24.20	31
OCaml	22.39	2.49	24.88	23
Rust	27.47	1.64	29.11	34
Haskell	49.03	3.63	52.66	31
Tcl	59.88	2.06	61.94	29
Fortran	73.52	0.10	73.62	57
BBawk	96.31	2.24	98.55	14

Figure 9-1: Run time and program size in various languages

authors of this book, who are not necessarily expert in the specific languages. Languages and especially compilers and libraries are moving targets, so this is a single experiment at a single point in time on a single computer.

All that said, however, it's clear that Awk is competitive, and some languages are surprisingly slower than adherents might have expected. The number of source lines is also interesting, because it makes clear that scripting languages, especially Awk, are more compact in expression. For instance, our Python version, shown in Figure 9-2, weighs in at 25 nonblank lines.

9.3 Conclusion

Awk is not a solution to every programming problem, but it's an indispensable part of a programmer's toolbox, especially on Unix, where easy connection of tools is a way of life. Although the larger examples in the book might give a different impression, most Awk programs are short and simple and do tasks the language was originally meant for: selecting information, counting, adding up numbers, and converting data from one form to another.

For tasks like these, where program development time is more important than run time, Awk is hard to beat. The implicit input loop and the pattern-action paradigm simplify and often entirely eliminate control flow. Field splitting parses the most common forms of input, while numbers and strings and the coercions between them handle the most common data

```
import sys, string
line = space = ""
def main():
    buf = sys.stdin.readline()
    while buf != "":
        if len(buf) == 1:
            printline()
            print("")
        else:
            for word in buf.split():
                addword(word)
        buf = sys.stdin.readline()
    printline()
def addword(word):
    global line, space
    if len(line) + len(word) > 60:
        printline()
    line = line + space + word
    space = " "
def printline():
    global line, space
    if len(line) > 0:
         print(line)
    line = space = ""
main()
```

Figure 9-2: Python version of formatter

types. Associative arrays provide both conventional array storage and the much richer possibilities of arbitrary subscripts. Regular expressions are a uniform notation for describing patterns of text. Default initialization and the absence of declarations shorten programs.

What we did not anticipate were the less conventional applications. For example, the transition from "not programming" to "programming" is gradual: the absence of the syntactic baggage of conventional languages like C or Java makes Awk easy enough to learn that it has been the first language for a surprising number of people.

In many cases, Awk is used to write a prototype, an experiment to demonstrate feasibility and to explore features and user interfaces, although sometimes the Awk program remains the production version. Awk has even been used for software engineering courses, because it's possible to experiment with designs much more readily than with larger languages.

Of course, one must be wary of going too far — any tool can be pushed beyond its limits — but many people have found Awk to be valuable for a wide range of problems. We hope we have suggested ways in which Awk might be useful to you as well.

You might find it interesting to compare Awk with similar languages. Among languages that were more or less contemporaneous with Awk in the 1970s, certainly the patriarch of the family is SNOBOL4, by Ralph Griswold, James Poage, and Ivan Polonsky. Although SNOBOL4

suffered from an unstructured input language, it was powerful and expressive. The REXX command interpreter language for IBM systems, by M. F. Cowlishaw, is another language in the same spirit, although with more emphasis on its role as a shell or command interpreter.

Today there are many more scripting languages. We've mentioned Perl, Python, and JavaScript, but to the list one should add PHP, Ruby, Lua, and Tcl, and perhaps functional languages like OCaml and Haskell. Today's shells are also significantly better as scripting languages on their own. There are plenty of choices now, and there will surely be more in the future.

Appendix A: Awk Reference Manual

This appendix explains, with examples, the constructs that make up Awk programs. Because it's a description of the complete language, the material is detailed, so we recommend that you skim it, then come back as necessary to check your understanding.

The first section describes patterns. The second section deals with actions: expressions, assignments, and control-flow statements. The remaining sections cover function definitions, output, input, and how Awk programs can call other programs.

Awk programs. The simplest Awk program is a sequence of pattern-action statements:

pattern { action }
pattern { action }

In some statements, the pattern may be missing; in others, the action and its enclosing braces may be missing. After Awk has checked your program to make sure there are no syntactic errors, it reads the input a line at a time, and for each input line, evaluates the patterns in order. For each pattern that matches the current input line, it executes the associated action. A missing pattern matches every input line, so every action with no pattern is performed on each line of input. A pattern-action statement consisting only of a pattern prints each input line matched by the pattern. The terms "input line" and "record" are used synonymously, though Awk also supports multiline records, where a record may contain several lines.

An Awk program is a sequence of pattern-action statements and function definitions. A function definition has the form

```
function name (parameter-list) { statements }
```

Pattern-action statements and function definitions are separated by newlines or semicolons and can be intermixed.

Statements are separated by newlines or semicolons or both.

The opening brace of an action must be on the same line as the pattern it accompanies; the remainder of the action, including the closing brace, may appear on the following lines.

Blank lines are ignored; they may be inserted before or after any statement to improve the readability of a program. Spaces and tabs may be inserted around operators and operands, again to enhance readability.

A semicolon by itself denotes the empty statement, as does { }.

A comment starts with the character # and ends at the end of the line, as in

{ print \$1, \$3 } # name and population

Comments may appear at the end of any line.

Backslashes may be used to break statements across multiple lines.

In addition, a statement may be broken without a backslash after a comma, left brace, &&, ||, do, else, and the closing right parenthesis in an if, for, or while statement.

A long statement may be spread over several lines by inserting a backslash and newline at each break:

{ print \
 \$1, # country name
 \$2, # area in thousands of square kilometers
 \$3 } # population in millions

As this illustrates, statements may be broken after commas, and a comment may be inserted at the end of each broken line.

In this book we have used several formatting styles, partly to illustrate different ones, and partly to keep programs short. For short programs, format doesn't much matter, but consistency and readability will help to keep longer programs manageable.

Commandlines. An Awk program is usually provided as a single argument on the commandline, or from a file named in a - f argument.

awk [-Fs] [-v var=value] 'program' optional list of filenames awk [-Fs] [-v var=value] -f progfile optional list of filenames

Multiple -f options are allowed; the Awk program is created by combining these files in order. If the filename is –, the program is read from the standard input.

The option -Fs sets the field separator variable FS to *s*.

The option --csv causes input to be treated as comma-separated values.

An option of the form -v var=value sets the variable *var* to *value* before the Awk program begins execution. Any number of -v arguments are permitted.

The option --version prints the version identification of the specific Awk program and terminates.

All options must appear before a literal *program*. The special optional argument -- marks the end of a list of optional arguments.

Command-line arguments are discussed further in Section A.5.5, below.

The Input File countries. As input for many of the Awk programs in the manual, we will use the countries file from Section 5.1. Each line contains the name of a country, its population in millions, its area in thousands of square kilometers, and the continent it is in. Values are from 2020; Russia has been arbitrarily placed in Europe. In the file, the four columns are separated by tabs; a single space separates North and South from America.

Russia	16376	145	Europe
China	9388	1411	Asia
USA	9147	331	North America
Brazil	8358	212	South America
India	2973	1380	Asia
Mexico	1943	128	North America
Indonesia	1811	273	Asia
Ethiopia	1100	114	Africa
Nigeria	910	206	Africa
Pakistan	770	220	Asia
Japan	364	126	Asia
Bangladesh	130	164	Asia

The file countries contains the following lines:

For the rest of the manual, the countries file is used when no input file is mentioned explicitly.

A.1 Patterns

Patterns control the execution of actions: when a pattern matches an input line, its associated action is executed. This section describes the types of patterns and the conditions under which they match.

Summary of Patterns

1. BEGIN { *statements* } The *statements* are executed once before any input has been read.

- END { statements } The statements are executed once after all input has been read.
- expression { statements } The statements are executed at each input line where the expression is true, that is, nonzero or non-null.
- 4. /regular expression / { statements } The statements are executed at each input line that contains a string matched by the regular expression.
- 5. *pattern*₁, *pattern*₂ { *statements* }

A range pattern matches each input line from a line matched by $pattern_1$ to the next line matched by $pattern_2$, inclusive; the *statements* are executed at each matching line. Both matches can occur on the same line.

BEGIN and END do not combine with other patterns, but there may be multiple instances. BEGIN and END always require an action; the *statements* and enclosing braces may be omitted from all other patterns.

A range pattern cannot be part of any other pattern.
A.1.1 BEGIN and END

The BEGIN and END patterns do not match any input lines. Rather, the statements in the BEGIN action are executed after Awk has processed the command line, but before it reads any input; the statements in the END action are executed after all input has been read. BEGIN and END thus provide a way to gain control for initialization and wrapup. BEGIN and END do not combine with other patterns. If there is more than one BEGIN, the associated actions are executed in the order in which they appear in the program; the same is true for multiple END patterns. Although it's not required, we put BEGIN first and END last.

One common use of a BEGIN action is to change the default way that input lines are split into fields. The field separator is controlled by a built-in variable called FS. By default, fields are separated by sequences of spaces and/or tabs; this behavior occurs when FS is set to a space.

If the command-line argument -csv is used, the input is treated as comma-separated values (CSV) format. Input fields are separated by commas, independent of the value of FS. Fields may be quoted with double-quote characters ". Such quoted fields may contain commas and double quotes, which are represented as ""; that is, two adjacent quotes are a literal quote. See Section A.5.2 for more details.

Setting FS to any character other than a space makes that character the field separator. A multi-character field separator is interpreted as a regular expression, as discussed below.

The following program uses the BEGIN action to set the field separator to a tab character (\t) and to put column headings on the output. The second printf statement, which is executed for each input line, formats the output into a table aligned under the column headings. The END action prints the totals. (Variables and expressions are discussed in Section A.2.1.)

```
# print countries with column headers and totals
BEGIN { FS = "\t" # make tab the field separator
    printf("%12s %6s %5s %s\n\n",
                         "COUNTRY", "AREA", "POP", "CONTINENT")
    }
    { printf("%12s %6d %5d %s\n", $1, $2, $3, $4)
        area += $2
        pop += $3
    }
END { printf("\n%12s %6d %5d\n", "TOTAL", area, pop) }
```

With the countries file as input, this program produces

COUNTRY	AREA	POP	CONTINENT
Russia	16376	145	Europe
China	9388	1411	Asia
USA	9147	331	North America
Brazil	8358	212	South America
India	2973	1380	Asia
Mexico	1943	128	North America
Indonesia	1811	273	Asia
Ethiopia	1100	114	Africa
Nigeria	910	206	Africa
Pakistan	770	220	Asia
Japan	364	126	Asia
Bangladesh	130	164	Asia
TOTAL	53270	4710	

A.1.2 Expression Patterns

Like most programming languages, Awk is rich in expressions for describing numeric computations, but it also has expressions for describing operations on strings. The term *string* means a sequence of zero or more characters represented in UTF-8. These may be stored in variables, or appear literally as string constants like "", "Asia", "[\Box # $\Lambda \dot{\tau}$ " and " \Box \Box ".

A *substring* is a contiguous sequence of zero or more characters within a string. The string "", which contains no characters, is called the *null* or *empty string*. In every string, the null string appears as a substring of length zero before the first character, between every pair of adjacent characters, and after the last character.

Any expression can be used as an operand of any operator. If an expression has a numeric value but an operator requires a string value, the numeric value is automatically transformed into a string; similarly, a string is converted into a number when an operator requires a numeric value. Type conversions and coercions are discussed in detail in Section A.2.2 below.

Any expression can be used as a pattern. If an expression used as a pattern has a nonzero or nonnull value at the current input line, then the pattern matches that line. The typical expression patterns are those involving comparisons between numbers or strings. A comparison expression contains one of the six relational operators, or one of the two string-matching operators \sim (tilde) and ! \sim that will be discussed in the next section. These operators are listed in Table A-1.

OPERATOR	MEANING
<	less than
<=	less than or equal to
==	equal to
! =	not equal to
>=	greater than or equal to
>	greater than
~	matched by
! ~	not matched by

TABLE A-1. COMPARISON OPERATORS

If the pattern is a comparison expression like NF > 10, then it matches the current input line when the condition is satisfied, that is, when the number of fields in the line is greater than 10. If the pattern is an arithmetic expression like NF, it matches the current input line when its numeric value is nonzero. If the pattern is a string expression, it matches the current input line when the string value of the expression is nonnull.

In a relational comparison, if both operands are numeric, a numeric comparison is made; otherwise, any numeric operand is converted to a string, and then the operands are compared as strings. The strings are compared character by character using UTF-8 ordering. One string is "less than" another if it would appear before the other according to this ordering, for example, "India" < "Indonesia" and "Asia" < "Asian". Comparisons are case-sensitive: "A" and "Z" both precede "a".

The pattern

\$3/\$2 > 0.5

selects lines where the value of the third field divided by the second is greater than 0.5, that is, where the population density is greater than 500 people per square kilometer, while

selects lines that begin with an M, N, O, etc.:

Russia	16376	145	Europe
USA	9147	331	North America
Mexico	1943	128	North America
Nigeria	910	206	Africa
Pakistan	770	220	Asia

Note that this also matches lines that begin with any character past M, which among other things includes any lower-case letter.

Sometimes the type of a comparison operator cannot be determined solely by the syntax of the expression in which it appears. The program

\$1 < \$4

could compare the first and fourth fields of each input line either as numbers or as strings. Here, the type of the comparison depends on the values of the fields, and it may vary from line to line. In the countries file, the first and fourth fields are always strings, so string comparisons are always made; the output is

Brazil	8358	212	South	America
Mexico	1943	128	North	America

As with all comparisons, the comparison is done numerically only if both fields are numbers; this would be the case with

\$2 < \$3

on the same data.

Section A.2.2 contains a complete discussion of strings, numbers, expressions, and coercions.

A compound pattern is an expression that combines other patterns, using parentheses and the logical operators || (OR), && (AND), and ! (NOT). A compound pattern matches the current input line if the expression evaluates to true. The following program uses the AND operator to select all lines in which the fourth field is Asia and the third field exceeds 500:

\$4 == "Asia" && \$3 > 500

The program

\$4 == "Asia" || \$4 == "Europe"

uses the OR operator to select lines with either Asia or Europe as the fourth field. As we will see in a moment, because the latter query is a test on string values, another way to write it is to use a regular expression with the alternation operator |:

\$4 ~ /^ (Asia|Europe) \$/

Two regular expressions are *equivalent* if they match the same strings. Test your understanding of the precedence rules for regular expressions: Are the two regular expressions ^Asia|Europe\$ and ^ (Asia|Europe) \$ equivalent?

If there are no occurrences of Asia or Europe in other fields, this pattern could also be written as

/Asia/ || /Europe/

or even

```
/Asia|Europe/
```

The || operator has the lowest precedence, then &&, and finally !. The && and || operators evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

A.1.3 Regular Expression Patterns

Awk provides a notation called *regular expressions* for specifying and matching strings of characters. Regular expressions are widely used throughout Unix, where restricted forms of regular expressions use "wild-card characters" for specifying sets of filenames. Regular expressions are also supported by text editors, and today are part of most programming languages, either directly in the syntax (as in Awk) or by libraries (as in Python).

A *regular expression pattern* tests whether a string contains a substring matched by a regular expression. In this section, we will discuss the most basic kinds of regular expressions and show how they appear in patterns; a detailed description of regular expressions follows in the next section.

Summary of Regular Expression Patterns

/regexpr/

Matches when the current input line contains a substring matched by regexpr.

expression ~ /regexpr/

Matches if the string value of expression contains a substring matched by regexpr.

expression !~ /regexpr/

Matches if the string value of expression does not contain a substring matched by regexpr.

Any expression may be used in place of /regexpr / in the context of ~ and !~. It is evaluated and then interpreted as a regular expression.

The simplest regular expression is a string of letters and numbers, like Asia, that matches itself. To turn a regular expression into a string-matching pattern, enclose it in slashes:

/Asia/

This pattern matches when the current input line contains the substring Asia, either as Asia by itself or as some part of a larger word like Asian or Pan-Asiatic. Note that spaces are significant within regular expressions: the string-matching pattern

/ Asia /

matches only when Asia is surrounded by spaces and thus matches no lines in countries.

The pattern above is one of three types of string-matching patterns. Its form is a regular expression r enclosed in slashes:

/r/

This pattern matches an input line if the line contains a substring matched by r.

The other two types of string-matching patterns use an explicit match operator:

expression ~ /r/ expression !~ /r/

The match operator ~ means "is matched by" and ! ~ means "is not matched by." The first pattern matches when the string value of *expression* contains a substring matched by the regular expression r; the second pattern matches if there is no such substring.

The left operand of a match operator is often a field: the pattern

\$4 ~ /Asia/

matches all input lines in which the fourth field contains Asia as a substring, while

\$4 !~ /Asia/

matches if the fourth field does not contain Asia anywhere.

Note that the string-matching pattern /Asia/ is a shorthand for \$0 ~ /Asia/.

A.1.4 Regular Expressions in Detail

A regular expression is a notation for specifying and matching strings. Like an arithmetic expression, a regular expression is a basic expression or one created by applying operators to

component expressions. To understand the strings matched by a regular expression, we need to understand the strings matched by its components.

Summary of Regular Expressions

The regular expression metacharacters are:

\ ^ \$. [] | () * + ? { }

A basic regular expression is one of the following:

a nonmetacharacter, such as A, that matches itself.

an escape sequence that matches a special symbol: e.g., \t matches a tab (see Table A-2).

a quoted metacharacter, such as $\ \$, that matches the metacharacter literally.

^, which matches the beginning of a string.

\$, which matches the end of a string.

., which matches any single character.

a character class: [ABC] matches any of the characters A, B, or C.

Character classes may include abbreviations: [0-9] matches any single digit, [A-Za-z] matches any single letter in either case, [[:class:]] matches any character in the class, which may be alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, or xdigit (hexadecimal digit).

Character classes may be complemented, to match any character not in the class: $[^0-9]$ matches any character except a digit; $[^[:cntrl:]]$ matches any non-control character.

These operators combine regular expressions into larger ones.

$r_1 r_2$	alternation: matches any string matched by r_1 or r_2
$r_1 r_2$	concatenation: matches xy where r_1 matches x and r_2 matches y
<i>r</i> *	matches zero or more consecutive strings matched by r
r+	matches one or more consecutive strings matched by r
r?	matches the null string or one string matched by r
$r\{m,n\}$	between m and n instances of r ; n is optional
(<i>r</i>)	grouping: matches the same strings as r

The operators are listed in order of increasing precedence. Redundant parentheses in regular expressions may be omitted as long as the precedence of operators is respected.

Metacharacters. Most characters in a regular expression match literal occurrences of themselves in the text, so a regular expression consisting of a single character like a letter or digit is a basic regular expression that matches itself.

However, the regular expression mechanism uses some characters to indicate a meaning other than their literal value. The characters

\ ^ \$. [] | () * + ? { }

are called *metacharacters* because they have special meanings as discussed below.

To preserve the literal meaning of a metacharacter in a regular expression, precede it by a backslash: the regular expression \$ matches the character \$. If a character is preceded by a single \$, we say that the character is *quoted*.

In a regular expression, an unquoted caret $^$ matches the beginning of a string, an unquoted dollar sign \$ matches the end of a string, and an unquoted period . matches any single character. Thus,

^C	matches a C at the beginning of a string; no special meaning elsewhere
C\$	matches a C at the end of a string; no special meaning elsewhere
^C\$	matches the string consisting of the single character C
^.\$	matches any string containing exactly one character
^\$	matches any string containing exactly three characters
	matches any three consecutive characters
\.\$	matches a period at the end of a string

Character classes. A regular expression consisting of a group of characters enclosed in brackets is called a *character class*; it matches any one of the enclosed characters. For example, [AEIOU] matches any of the characters A, E, I, O, or U.

Ranges of characters can be abbreviated in a character class by using a hyphen. The character immediately to the left of the hyphen defines the beginning of the range; the character immediately to the right defines the end. Thus, [0-9] matches any digit, and [a-zA-Z][0-9] matches a letter followed by a digit. Without both a left and right operand, a hyphen in a character class denotes itself, so the character classes [+-] and [-+] match either a + or a -. The character class [A-Za-z-]+ matches words that include hyphens.

Ranges of Unicode characters work so long as the range is of manageable size, roughly 256 characters. In general, if the character set in question fits on a single page in the Unicode descriptions at unicode.org, a range will work; for example, the character class [r-f] matches Japanese Katakana characters.

Special character classes like [:alpha:] match any one of a range of characters defined by the local environment, as set by the LOCALE shell variable. This enables some languageindependent character-class matching. For example, if the locale is set to the value LC_ALL=fr_FR.UTF-8, then the regular expression [[:alpha:]] matches accented letters like é and à , while in the locale en_EN, it does not.

Complemented character classes. A complemented character class is one in which the first character after the [is a $^$. Such a class matches any character *not* in the group following the caret. Thus, [$^0-9$] matches any character except a digit; [$^a-zA-Z$] matches any character except an upper or lower-case letter.

^[ABC]	matches an A, B, or C at the beginning of a string
^[^ABC]	matches any character at the beginning of a string, except A, B, or C
[^ABC]	matches any character other than an A, B, or C
^[^a-z]\$	matches any single-character string, except a lower-case letter
^[^[:lower:]]\$	also matches any single-character string, except a lower-case letter

Inside a character class, all characters have their literal meaning, except for the quoting character \backslash , ^ at the beginning, and – between two characters. Thus, [.] matches a period and ^ [^^] matches any character except a caret at the beginning of a string.

Grouping. Parentheses are used in regular expressions to specify how components are grouped. There are two binary regular expression operators: alternation and concatenation. The alternation operator | is used to specify alternatives: if r_1 and r_2 are regular expressions, then $r_1 | r_2$ matches any string matched by r_1 or by r_2 .

There is no explicit concatenation operator. If r_1 and r_2 are regular expressions, then $(r_1) (r_2)$ (with no space between (r_1) and (r_2)) matches any string of the form xy where r_1 matches x and r_2 matches y. The parentheses around r_1 or r_2 can be omitted if the contained regular expression does not contain the alternation operator. The regular expression

```
(Asian|European|North American) (male|female) (black|blue)bird matches twelve strings ranging from
```

```
Asian male blackbird
```

to

North American female bluebird

Repetitions. The symbols *, +, and ? are unary operators used to specify repetitions in regular expressions. If r is a regular expression, then (r) * matches any string consisting of zero or more consecutive substrings matched by r; (r) + matches any string consisting of one or more consecutive substrings matched by r; and (r)? matches zero or one instances of r, that is the null string or any string matched by r.

The notation (r) $\{m, n\}$ specifies a match of between m and n (inclusive) occurrences of the preceding regular expression; if , n is omitted, the pattern matches exactly m occurrences.

If r is a basic regular expression, parentheses can be omitted.

В*	matches the null string or B or BB, and so on
AB*C	matches AC or ABC or ABBC, and so on
AB+C	matches ABC or ABBC or ABBBC, and so on
ABB*C	also matches ABC or ABBC or ABBBC, and so on
AB?C	matches AC or ABC
[A-Z]+	matches any string of one or more upper-case letters
(AB) +C	matches ABC, ABABC, ABABABC, and so on
X(AB){1,2}Y	matches XABY, XABABY, but not XABABABY and so on

In regular expressions, the alternation operator | has the lowest precedence, then concatenation, and finally the repetition operators *, +, ? and { }. As in arithmetic expressions, operators of higher precedence are evaluated before lower ones. These conventions allow parentheses to be omitted: ab|cd is the same as (ab)|(cd), and ab|cd*e\$ is the same as (ab)|(c(d*)e\$).

Escapes in regular expressions and strings. Within regular expressions and strings, Awk uses certain character sequences, called *escape sequences*, to specify characters for which there may be no other notation. For example, \n stands for a newline character, which cannot otherwise appear in a string or regular expression; \b stands for backspace; \t stands for tab; and $\/$ represents a slash. Any arbitrary value can be entered with an octal or hexadecimal escape: $\033$ and $\0x1b$ both represent the ASCII escape character. An arbitrary Unicode character can be entered as $\uh...$, where h... is a sequence of up to 8 hexadecimal digits that represent a valid Unicode character; for example, the character (:) is $\ulf642$.

It's important to note that such escape sequences have special meaning only within an Awk *program*; in data, they are just characters. The complete list of escape sequences is shown in Table A-2.

Examples. To finish our discussion of regular expressions, here are some examples of useful string-matching patterns containing regular expressions with unary and binary operators,

SEQUENCE	MEANING
∖a	alarm (bell)
∖b	backspace
\f	formfeed
∖n	newline (line feed)
\r	carriage return
\t	tab
\v	vertical tab
$\setminus ddd$	octal value <i>ddd</i> ; <i>ddd</i> is 1 to 3 digits between 0 and 7
\xhh	hexadecimal value; hh is 1 or 2 hexadecimal digits in upper or lower case
∖u <i>h</i>	Unicode value; h is up to 8 hexadecimal digits in upper or lower case
$\setminus c$	any other character c literally, e.g., $\$ " for " and $\ \$ for $\$

 TABLE A-2.
 ESCAPE SEQUENCES

along with a description of the kinds of input lines they match. Recall that a string-matching pattern /r/ matches the current input line if the line contains at least one substring matched by r.

```
/^[0-9]+$/
       matches any input line that consists of one or more decimal digits
/^[0-9][0-9][0-9]$/
       exactly three digits
/^[0-9]{3}$/
       also exactly three digits
/^(\+|-)?[0-9]+\.?[0-9]*$/
       a decimal number with an optional sign and optional fraction
/^[+-]?[0-9]+[.]?[0-9]*$/
       also a decimal number with an optional sign and optional fraction
/^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)([eE][+-]?[0-9]+)?$/
       a floating point number with optional sign and optional exponent
/^[A-Za-z_][A-Za-z_0-9]*$/
       a letter or underscore followed by any letters, underscores, or digits (e.g., a variable name)
/^[A-Za-z]$|^[A-Za-z][0-9]$/
       a letter or a letter followed by a digit
/^[A-Za-z][0-9]?$/
       also a letter or a letter followed by a digit
```

Since + and . are metacharacters, they have to be preceded by backslashes in the fourth example to match literal occurrences. These backslashes are not needed within character classes, so the fifth example shows an alternate way to describe the same numbers.

Any regular expression enclosed in slashes can be used as the right-hand operand of a matching operator: the program

\$2 !~ /^[0-9]+\$/

prints all lines in which the second field is not a string of digits.

Table A-3 summarizes regular expressions and the strings they match. The operators are listed in order of increasing precedence. Characters are Unicode code points.

EXPRESSION	MATCHES
С	the nonmetacharacter c
$\setminus c$	escape sequence or literal character c
^	beginning of string
\$	end of string
	any character
$[c_1 c_2 \dots]$	any character in $c_1 c_2 \dots$
[^ <i>c</i> ₁ <i>c</i> ₂]	any character not in $c_1 c_2 \dots$
$[c_1 - c_2]$	any character in the range beginning with c_1 and ending with c_2
$[^{c_1-c_2}]$	any character not in the range c_1 to c_2
<i>r</i> ₁ <i>r</i> ₂	any string matched by r_1 or r_2
$(r_1)(r_2)$	any string xy where r_1 matches x and r_2 matches y;
	parentheses are not needed around subexpressions with no alternations
(<i>r</i>)*	zero or more consecutive strings matched by r
(r)+	one or more consecutive strings matched by r
(r)?	zero or one string matched by r
$(r) \{ m, n \}$	m through n consecutive strings matched by r ; n may be omitted;
	parentheses are not needed around basic regular expressions
(<i>r</i>)	any string matched by r

A.1.5 Range Patterns

A range pattern consists of two patterns separated by a comma, as in

 pat_1 , pat_2

A range pattern matches each line between an occurrence of pat_1 and the next occurrence of pat_2 inclusive; pat_2 may match the same line as pat_1 , making the range a single line.

Matching begins whenever the first pattern of a range matches; if no instance of the second pattern is subsequently found, then all lines to the end of the input are matched:

```
/Europe/, /Africa/
```

prints

Russia	16376	145	Europe
China	9388	1411	Asia
USA	9147	331	North America
Brazil	8358	212	South America
India	2973	1380	Asia
Mexico	1943	128	North America
Indonesia	1811	273	Asia
Ethiopia	1100	114	Africa

FNR is the number of the line just read from the current input file and FILENAME is the filename itself; both are built-in variables. Thus, the program

FNR == 1, FNR == 5 { print FILENAME ": " \$0 }

prints the first five lines of each input file with the filename prefixed. Alternatively, this program could be written as

FNR <= 5 { print FILENAME ": " \$0 }</pre>

A range pattern cannot be part of any other pattern.

A.2 Actions

In a pattern-action statement, the pattern determines when the action is executed. Sometimes an action is simple: a single print or assignment. Other times, it may be a sequence of several statements separated by newlines or semicolons. This section begins the description of actions by discussing expressions and control-flow statements. The following sections present user-defined functions, and statements for input and output.

Summary of Actions

The statements in actions can include:

expressions, with constants, variables, assignments, function calls, etc.

```
print expression-list
printf(format, expression-list)
if (expression) statement
if (expression) statement else statement
while (expression) statement
for (expression; expression; expression) statement
for (variable in array) statement
do statement while (expression)
break
continue
next
nextfile
exit
exit expression
{ statements }
```

A.2.1 Expressions

We begin with expressions, because expressions are the simplest statements, and most other statements are made up of expressions of various kinds. An expression is formed by combining primary expressions and other expressions with operators. The primary expressions are the primitive building blocks: they include constants, variables, array references, function invocations, and various built-ins, like field names.

Our discussion of expressions starts with constants and variables. Then come the operators that can be used to combine expressions. These operators fall into five categories: arithmetic, comparison, logical, conditional, and assignment. The built-in arithmetic and string functions come next, followed at the end of the section by the description of arrays. **Constants.** There are two types of constants, string and numeric. A string constant is created by enclosing a sequence of characters in quotation marks, as in "hello, world" or "Asia" or "". String constants may contain the escape sequences listed in Table A-2. A long string can be split into multiple lines with backslashes:

```
s = "a really very long \
string split over two lines"
```

The newline that follows the backslash is removed; it is not part of the string, so the result is equivalent to

s = "a really very long string split over two lines"

Any spaces at the beginning of the continuation are included.

A numeric constant can be an integer like 1127, a decimal number like 3.14, or a number in scientific (exponential) notation like 6.022E+23. Different representations of the same number have the same numeric value: the numbers 1e6, 1.00E6, 10e5, 0.1e7, and 1000000 are numerically equal.

All numbers are stored in double-precision floating point, the precision of which is machine dependent, though usually about 15 decimal digits.

Two special numeric values are recognized, "+nan" and "+inf", which represent NaN, the "not a number" value, and infinity. These must include an explicit + or - sign both as literals in a program and as data input.

The names are not case sensitive, so NaN and Inf are also valid.

The nan and inf values can be generated by arithmetic expressions; for example.

```
$ awk '{print " $1/$2}'
12
   0.5
1 +nan
   +nan
+nan 1
  +nan
+nan +nan
  +nan
+nan -inf
  +nan
+inf +inf
   -nan
0 +inf
   0
+inf 0
awk: division by zero
 input record number 7, file
source line number 1
```

Variables. Expressions can contain several kinds of variables: user-defined, built-in, and fields. Variable names are sequences of letters, digits, and underscores that do not begin with a digit; all built-in variables have all-upper-case names.

A variable has a value that is a string or a number or both. Since the type of a variable is not declared, Awk infers the type from context. When necessary, Awk will convert a string value into a numeric one, or vice versa. For example, in \$4 == "Asia" { print \$1, 1000 * \$2 }

\$2 is converted into a number if it is not one already, and \$1 and \$4 are converted into strings if they are not already.

An uninitialized variable has the string value "" (the null string) and the numeric value 0.

Built-In Variables. Table A-4 lists the built-in variables. These variables can be used in all expressions, and may be reset by the user. FILENAME is set each time a new file is read. FNR, NF, and NR are set each time a new record is read; additionally, NF is reset when \$0 changes or when a new field is created. Conversely, if NF changes, \$0 is recomputed when its value is needed. The variables RLENGTH and RSTART change as a result of invoking the match function.

VARIABLE	MEANING	DEFAULT
ARGC	number of command-line arguments, including command name	-
ARGV	array of command-line arguments, numbered 0ARGC-1	-
CONVFMT	conversion format for numbers	"%.6g"
ENVIRON	array of shell environment variables	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	"%.6g"
OFS	output field separator for print	
ORS	output record separator for print	"\n"
RLENGTH	length of string matched by match function	-
RS	input record separator	"\n"
RSTART	start of string matched by match function	-
SUBSEP	subscript separator	"\034"

TABLE A-4. BUILT-IN VARIABLES

Field Variables. The fields of the current input line are called \$1, \$2, through \$NF; \$0 refers to the whole line. Fields share the properties of other variables — they may be used in arithmetic or string operations, and they may be assigned to. Thus one can divide the second field in each line of countries by 1000 to express areas in millions of square kilometers instead of thousands:

{ \$2 = \$2 / 1000; print }

One can assign a new string to a field:

```
BEGIN { FS = OFS = "\t" }
$4 == "North America" { $4 = "NA" }
$4 == "South America" { $4 = "SA" }
{ print }
```

In this program, the BEGIN action sets FS, the variable that controls the input field separator, and OFS, the output field separator, both to a tab. The print statement in the fourth line

prints the value of 0 after it has been modified by previous assignments. When 0 is changed by assignment or substitution, 1, 2, etc., and NF will all be recomputed; likewise, when one of 1, 2, etc., is changed, 0 is reconstructed using OFS to separate fields.

Fields can also be specified by expressions. For example, (NF-1) is the next-to-last field of the current line. The parentheses are needed: NF-1 is one less than the numeric value of the last field.

A field variable referring to a nonexistent field, e.g., (NF+1), has as its initial value the null string. A new field can be created by assigning a value to it. For example, the following program creates a fifth field containing the population density:

```
BEGIN { FS = OFS = "\t" }
{ $5 = 1000 * $3 / $2; print }
```

Any intervening fields are created when necessary and given null values.

The number of fields can vary from line to line.

Summary of Expressions

The primary expressions are:

numeric and string constants, variables, fields, function calls, array elements.

These operators combine expressions:

```
assignment operators = += -= *= /= &= ^=
conditional expression operator ?:
logical operators | | (OR), && (AND), ! (NOT)
matching operators ~ and !~
relational operators < <= == != > >=
concatenation (no explicit operator)
arithmetic operators + - * / & ^
unary + and -
increment and decrement operators ++ and -- (prefix and postfix)
parentheses for grouping
```

Assignment Operators. There are seven assignment operators that can be used in expressions called *assignments*. The simplest assignment is an expression of the form

var = expr

where *var* is a variable or field name, and *expr* is any expression. For example, to compute the total population and number of Asian countries, we could write

```
$4 == "Asia" { pop = pop + $3; n = n + 1 }
END { print "Total population of the", n,
                                "Asian countries is", pop, "million."
}
```

Applied to countries, the program produces

Total population of the 6 Asian countries is 3574 million.

The first action contains two assignments, one to accumulate population, and the other to

count countries. The variables are not explicitly initialized, yet everything works properly because each variable is initialized by default to the string value "" and the numeric value 0.

We also use default initialization to advantage in the following program, which finds the country with the largest population:

The result:

country with largest population: China 1411

Note, however, that this program is correct only when at least one value of \$3 is positive.

The other six assignment operators are +=, -=, *=, /=, &=, and $^=$. Their meanings are similar: $v \ op = e$ has the same effect as $v = v \ op \ e$. The assignment

pop = pop + \$3

can be written more concisely using the assignment operator +=:

pop += \$3

This statement has the same effect as the longer version — the variable on the left is incremented by the value of the expression on the right — but += is more compact and often clearer. In addition, v is evaluated only once so a complicated computation like

```
v[substr($0, index($0, "!")+1)] += 2
```

will run faster.

As another example,

{ \$2 /= 1000; print }

divides the second field by 1000, then prints the line.

An assignment is an expression; its value is the new value of the left side. Thus assignments can be used inside any expression. In the multiple assignment

 $FS = OFS = " \setminus t"$

both the field separator and the output field separator are set to tab. Assignment expressions are also common within tests, such as:

if $((n = length(\$0)) > 0) \dots$

though this kind of use can be confusing. Don't forget the parentheses.

Conditional Expression Operator. A conditional expression has the form

 $expr_1$? $expr_2$: $expr_3$

First, $expr_1$ is evaluated. If it is true, that is, nonzero or nonnull, the value of the conditional expression is the value of $expr_2$; otherwise, it is the value of $expr_3$. Only one of $expr_2$ and $expr_3$ is evaluated.

The following program uses a conditional expression to print the reciprocal of \$1, or a warning if \$1 is zero:

{ print (\$1 != 0 ? 1/\$1 : "\$1 is zero, line " NR) }

As with nested assignments, conditional expressions can be abused to create inscrutable code.

Logical Operators. The logical operators && (AND), || (OR), and ! (NOT) are used to create logical expressions by combining other expressions. A logical expression has the value 1 if it is true and 0 if false. In the evaluation of a logical operator, an operand with a nonzero or nonnull value is treated as true; other values are treated as false. The operands of expressions separated by && or || are evaluated from left to right, and evaluation ceases as soon as the value of the complete expression can be determined. This means that in

expr₁ && expr₂

 $expr_2$ is not evaluated if $expr_1$ is false, while in

 $expr_3 \mid \mid expr_4$

 $expr_4$ is not evaluated if $expr_3$ is true.

Newlines may be inserted after the && and || operators. The precedence of && is higher than ||, so an expression like

A & & B | | C & & D

is parsed as

(A & & B) || (C & & D)

Parentheses should be used to make sure such expressions are clear to the reader.

Relational Operators. Relational or comparison expressions are those containing either a relational operator or a regular expression matching operator. The relational operators are <, <=, == (equals), != (not equals), >=, and >. The regular expression matching operators are ~ (is matched by) and $! \sim$ (is not matched by).

The value of a comparison expression is 1 if it is true and 0 otherwise. Similarly, the value of a matching expression is 1 if true, 0 if false, so

\$4 ~ /Asia/

is 1 if the fourth field of the current line contains Asia as a substring, or 0 if it does not.

Arithmetic Operators. Awk provides the usual +, -, *, /, \$, and $^$ arithmetic operators. The \$ operator computes remainders: x\$y is the remainder when x is divided by y; its behavior depends on the particular computer if x or y is negative. The $^$ operator is exponentiation: x^y is x^y . Note that $^$ has a different meaning (bitwise exclusive OR) in C and many other languages.

All arithmetic is done in double-precision floating point, which is usually about 15 decimal digits.

Unary Operators. The unary operators are + and –, with the obvious meanings.

Increment and Decrement Operators. The assignment n = n + 1 is usually written ++n or n++ using the unary increment operator ++, which adds 1 to a variable. The prefix form ++n increments n before delivering its value; the postfix form n++ increments n after delivering its value. This makes a difference when ++ is used in an assignment. If n is initially 1, then the assignment i = ++n increments n and assigns the new value 2 to i, while the assignment i = n++ increments n but assigns the old value 1 to i. To just increment n, however,

there's no difference between n++ and ++n. The prefix and postfix decrement operator --, which subtracts 1 from a variable, works the same way.

Built-In Arithmetic Functions. The built-in arithmetic functions are shown in Table A-5. These functions can be used as primary expressions in all expressions. In the table, x and y are arbitrary expressions.

FUNCTION	VALUE RETURNED
atan2(y, x)	arctangent of y/x in the range $-\pi$ to π
$\cos(x)$	cosine of x, with x in radians
$\exp(x)$	exponential function of x , e^x
int(x)	integer part of x; truncated towards 0
log(x)	natural (base e) logarithm of x
rand()	random number r, where $0 \le r \le 1$
sin(x)	sine of x, with x in radians
sqrt(x)	square root of x
<pre>srand(x)</pre>	x is new seed for rand(); use time of day if x is omitted; return previous seed

TABLE A-5. BUILT-IN ARITHMETIC FUNCTIONS

Useful constants can be computed with these functions: atan2(0, -1) gives π and exp(1) gives e, the base of the natural logarithms. To compute the base-10 logarithm of x, use log(x)/log(10).

The function rand() returns a pseudo-random floating point number greater than or equal to 0 and less than 1. Calling srand(x) sets the starting seed of the generator from x and returns the previous seed. Calling srand() sets the starting point from the time of day. If srand is not called, rand starts with the same value each time the program is run.

The assignment

```
randint = int(n * rand()) + 1
```

sets randint to a random integer between 1 and n inclusive, using the int function to discard the fractional part. The assignment

x = int(x + 0.5)

rounds the value of x to the nearest integer when x is positive.

String Operators. There is only one string operation, concatenation. It has no explicit operator: string expressions are created by writing constants, variables, fields, array elements, function values, and other expressions next to one another. The program

```
{ print NR ":" $0 }
```

prints each line preceded by its line number and a colon, with no spaces. The number NR is converted to its string value (and so is \$0 if necessary); then the three strings are concatenated and the result is printed.

Strings as Regular Expressions. So far, in all of our examples of matching expressions, the right-hand operand of \sim and ! \sim has been a regular expression enclosed in slashes. But in fact any expression can be used as the right operand of these operators. Awk evaluates the

expression, converts the value to a string if necessary, and interprets the string as a regular expression. For example, the program

```
BEGIN { digits = "^[0-9]+$" }
$2 ~ digits
```

will print all lines in which the second field is a string of digits.

Since expressions can be concatenated, a regular expression can be built up from components. The following program echoes input lines that are valid floating point numbers:

```
BEGIN {
    sign = "[+-]?"
    decimal = "[0-9]+[.]?[0-9]*"
    fraction = "[.][0-9]+"
    exponent = "([eE]" sign "[0-9]+)?"
    number = "^" sign "(" decimal "|" fraction ")" exponent "$"
}
$0 ~ number
```

In a matching expression, a quoted string like $"^[0-9]+\$"$ can normally be used interchangeably with a regular expression enclosed in slashes, such as $/^[0-9]+\$/$. There is one exception, however. If the string in quotes is to match a literal occurrence of a regular expression metacharacter, one extra backslash is needed to protect the protecting backslash itself. That is,

\$0 ~ / (\+|-) [0-9]+/

and

\$0 ~ "(\\+|-)[0-9]+"

are equivalent.

This behavior may seem arcane, but it arises because one level of protecting backslashes is removed when a quoted string in a program is parsed by Awk. If a backslash is needed in front of a metacharacter to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string. If the right operand of a matching operator is a variable or field, as in

x ~ \$1

then the additional level of backslashes is not needed in the first field because backslashes have no special meaning in data.

As an aside, it's easy to test your understanding of regular expressions interactively: the program

\$1 ~ \$2

lets you type in a string and a regular expression; it echoes the line back if the string matches the regular expression.

Built-In String Functions. Awk provides the built-in string functions shown in Table A-6. In this table, r represents a regular expression (either as a string or enclosed in slashes), s and t are string expressions, and n and p are integers. Strings are represented as UTF-8 characters.

The arguments of a function call are all evaluated before the function is called, but the order of evaluation is unspecified.

FUNCTION	DESCRIPTION
gsub (r, s)	substitute <i>s</i> for <i>r</i> globally in $\$0$,
	return number of substitutions made
gsub(<i>r</i> , <i>s</i> , <i>t</i>)	substitute s for r globally in string t ,
	return number of substitutions made
index(s,t)	return first position of string t in s , or 0 if t is not present
length(s)	return number of Unicode characters in s;
	return number of elements if s is an array
match(s, r)	test whether s contains a substring matched by r ;
	return index or 0; sets RSTART and RLENGTH
<pre>split(s, a)</pre>	split s into array a on FS or as CSV if $-csv$ is set,
	return number of elements in a
<pre>split(s, a, fs)</pre>	split s into array a on field separator fs,
	return number of elements in a
<pre>sprintf(fmt, expr-list)</pre>	return expr-list formatted according to format string fmt
sub(<i>r</i> , <i>s</i>)	substitute <i>s</i> for the leftmost longest substring of 0 matched by <i>r</i> ;
	return number of substitutions made
sub(<i>r</i> , <i>s</i> , <i>t</i>)	substitute s for the leftmost longest substring of t matched by r ;
	return number of substitutions made
substr(s, p)	return suffix of s starting at position p
substr(s, p, n)	return substring of s of length at most n starting at position p
tolower(s)	return s with upper case ASCII letters mapped to lower case
toupper(s)	return s with lower case ASCII letters mapped to upper case

TABLE A-6. BUILT-IN STRING FUNCTIONS

The function index(s, t) returns the leftmost position where the string t begins in s, or zero if t does not occur in s. The first character in a string is at position 1, so

```
index("banana", "an")
```

returns 2.

The function match(s, r) finds the leftmost longest substring in the string s that is matched by the regular expression r. It returns the index where the substring begins or 0 if there is no matching substring. It also sets the built-in variables RSTART to this index and RLENGTH to the length of the matched substring.

The function split(s, a, fs) splits the string s into the array a according to the separator fs and returns the number of elements. It is described after arrays, at the end of this section.

The string function $\operatorname{sprintf}(format, expr_1, expr_2, \ldots, expr_n)$ returns (without printing) a string containing $expr_1, expr_2, \ldots, expr_n$ formatted according to the printf specifications in the string value of the expression format. Thus, the statement

x = sprintf("%10s %6d", \$1, \$2)

assigns to x the string produced by formatting the values of \$1 and \$2 as a ten-character string and a decimal number in a field of width at least six. Section A.4.3 contains a complete

description of the printf format-conversion characters.

The functions sub and gsub are patterned after the substitute command in the Unix text editor ed. The function sub (r, s, t) first finds the leftmost longest substring matched by the regular expression r in the target string t, which must be a variable, field, or array element; it then replaces the substring by the substitution string s. As in most text editors, "leftmost longest" means that the leftmost match (that is, the first match) is found first, then extended as far as possible.

In the target string banana, for example, anan is the leftmost longest substring matched by the regular expression (an) +. By contrast, the leftmost longest match of (an) * is the null string before b, which may be surprising when first encountered.

The sub function returns the number of substitutions made, which will be zero or one. The function sub (r, s) is a synonym for sub (r, s, \$0).

The function gsub (r, s, t) is similar, except that it successively replaces the leftmost longest nonoverlapping substrings matched by r with s in t; it returns the number of substitutions made. (The "g" is for "global," meaning everywhere.) For example, the program

{ gsub(/USA/, "United States"); print }

will transcribe its input, replacing all occurrences of "USA" by "United States". (In such examples, when \$0 changes, the fields and NF change too.) And

```
b = "banana"
gsub(/ana/, "anda", b)
```

will replace banana by bandana in b; matches are nonoverlapping.

In a substitution performed by either sub (r, s, t) or gsub (r, s, t), any occurrence of the character & in s will be replaced by the substring matched by r. Thus

b = "banana"
gsub(/a/, "aba", b)

replaces banana by babanabanaba in b; so does

gsub(/a/, "&b&", b)

The special meaning of & in the substitution string can be turned off by preceding it with a backslash, as in $\$

The function substr(s, p) returns the suffix of s that begins at position p. If substr(s, p, n) is used, only the first n characters of the suffix are returned; if the suffix is shorter than n, then the entire suffix is returned. For example, we could abbreviate the country names in countries to their first six characters by the program

{ \$1 = substr(\$1, 1, 6); print \$0 }

to produce

```
Russia 16376 145 Europe
China 9388 1411 Asia
USA 9147 331 North America
Brazil 8358 212 South America
India 2973 1380 Asia
Mexico 1943 128 North America
Indone 1811 273 Asia
Ethiop 1100 114 Africa
Nigeri 910 206 Africa
Pakist 770 220 Asia
Japan 364 126 Asia
Bangla 130 164 Asia
```

Setting \$1 (or any other field) forces Awk to recompute \$0 and thus the fields are now separated by a space (the default value of OFS), no longer by a tab.

Strings are concatenated merely by writing them one after another in an expression. For example, on the countries file,

```
/Asia/ { s = s $1 " " }
END { print s }
```

prints

China India Indonesia Pakistan Japan Bangladesh

by building s up a piece at a time starting with an initially empty string. To remove the extra space at the end, you could use

```
print substr(s, 1, length(s)-1)
```

instead of print s in the END action.

A.2.2 Type Conversions

Each Awk variable and field can potentially hold a string value, a numeric value, or both, at any time. This section sets out the rules for how string and numeric values are treated in assignments, comparisons, expression evaluation, input, and output.

Assignments. When a variable is set by an assignment

var = expr

its type is set to that of the expression. ("Assignment" includes the assignment operators +=, -=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in v1 = v2, then the type of v1 is set to that of v2.

Number or String? The value of an expression may be automatically converted from a number to a string or vice versa, depending on what operation is applied to it. In an arithmetic expression like

pop + \$3

the operands pop and \$3 must be numeric, so their values will be forced or *coerced* to numbers if they are not already. Similarly, in the assignment expression

pop += \$3

pop and \$3 must be numbers, so after the expression is evaluated, pop will be numeric and \$3 will have a numeric value, which may have been computed from its string value if it had one. In a string expression like

\$1 \$2

the operands \$1 and \$2 must be strings to be concatenated, so they will be coerced to strings if necessary; if they had numeric values, those will be unchanged.

The type of a field is determined by context when possible; for example,

\$1++

implies that \$1 must be coerced to numeric if necessary, and

\$1 = \$1 "," \$2

implies that \$1 and \$2 must be coerced to strings if necessary.

Comparisons and coercions. In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on the string values.

Uninitialized variables have the numeric value 0 and the string value "". Accordingly, if x is uninitialized,

if (x) ...

is false, and

if (!x) ... if (x == 0) ... if (x == "") ...

are all true because x is both 0 and "". But if x is uninitialized,

if (x == "0") ...

is false because x is "", which is a string value, not numeric.

There are two idioms for coercing an expression of one type to the other:

number "" concatenate a null string to *number* to coerce it to a string string + 0 add zero to string to coerce it to a number

Thus, to force a string comparison between two fields, coerce one field to string:

\$1 "" == \$2

To force a numeric comparison, coerce *both* fields to numeric:

\$1 + 0 == \$2 + 0

This works regardless of what the fields contain.

Type inference. In contexts where types cannot be reliably determined, such as

if (\$1 == \$2) ...

the type of each field is determined heuristically on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value ""; they are not numeric. Non-existent fields (i.e., fields past NF) and \$0 for blank lines are treated this way too.

Let us examine the meaning of a comparison like

\$1 == \$2

that involves fields. Here, the type of the comparison depends on whether the fields contain numbers or strings, and this can only be determined when the program runs; the type of the comparison may differ from input line to input line. When Awk creates a field at run time, it automatically sets its type to string; in addition, if the field contains a valid number, it also gives the field a numeric type.

For example, the comparison 1 = 2 will be numeric and succeed if 1 and 2 have any of the values

1 1.0 +1 1e0 0.1e+1 10E-1 001

because all these values are different representations of the numeric value 1. However, this same expression will be a string comparison and hence fail on each of these pairs:

0	(null)
0.0	(null)
0	0 x
1e5000	1.0e5000

In the first three pairs, the second field is not a number. The last pair will be compared as strings on computers where the values are too large to be represented as numbers.

As it is for fields, so it is for array elements created by split.

Mentioning an array element in an expression causes it to exist, with the values 0 and "" as described above. Thus, if arr[i] does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value "" and thus the if is satisfied.

This property leads to an elegant program for eliminating duplicate records from an input stream:

```
!a[$0]++ # equivalently, a[$0]++ == 0
```

counts the number of times any particular line appears, but prints only the first occurrence, because that is the only time when the count for the specific array element is zero.

The test

```
if (i in arr) ...
```

determines if arr[i] exists without the side effect of creating it.

Number to string conversions. The print statement

print \$1

prints the string value of the first field; thus, the output is identical to the input.

Nonexistent fields and fields that are explicitly null have only the string value ""; they are not numeric, but when coerced to numbers they acquire the numeric value 0. Array subscripts are always strings; a numeric subscript is converted to its string value.

The numeric value of a string is the value of the longest numeric prefix of the string. Thus

BEGIN { print "1E2"+0, "12E"+0, "E12"+0, "1X2Y3"+0 }

yields

100 12 0 1

For printing, the string value of a number is computed by formatting the number with the output format conversion OFMT; its default value is "%.6g". Thus

BEGIN { print 1E2, 12E-2, E12 "", 1.23456789 }

gives

```
100 0.12 1.23457
```

Look carefully at this output: there is an empty field corresponding to the third argument, E12 "".

For other conversions from number to string, the string value of a number is computed by formatting the number with the conversion format conversion CONVFMT.

CONVFMT controls the conversion of numeric values to strings for concatenation, comparison, and creation of array subscripts. The default value of CONVFMT is also "%.6g". The default values of OFMT and CONVFMT can be changed by assigning them new values. If CONVFMT were changed to "%.2f", for example, coerced numbers would be compared with two digits after the decimal point. In both cases, integral values convert to integers regardless of CONVFMT and OFMT.

Summary of Operators. The operators that can appear in expressions are summarized in Table A-7. Expressions can be created by applying these operators to constants, variables, field names, array elements, function results, and other expressions.

The operators are listed in order of increasing precedence. Operators of higher precedence are evaluated before lower ones; this means, for example, that * is evaluated before + in an expression. All operators are left associative except the assignment operators, the conditional operator, and exponentiation, which are right associative. Left associativity means that operators of the same precedence are evaluated left to right; thus 3-2-1 is (3-2)-1, not 3-(2-1).

Since there is no explicit operator for concatenation, it is wise to parenthesize expressions involving other operators in concatenations. Consider the program

1 < 0{ print "abs(\$1) = " -\$1 }

The expression following print seems to use concatenation, but is actually a subtraction. The programs

```
1 < 0 { print "abs($1) = " (-$1) }
```

and

```
1 < 0 { print "abs($1) =", -$1 }
```

both do what was intended.

A.2.3 Control-Flow Statements

Awk provides braces for grouping statements, an if-else statement for decision-making, and while, for, and do statements for looping. All of these statements were adopted

OPERATION	OPERATORS	EXAMPLE	MEANING OF EXAMPLE
assignment	= += -= *=	x *= 2	x = x * 2
	/= %= ^=		
conditional	?:	x ? y : z	if x is true then y else z
logical OR	11	х у	1 if x or y is true,
			0 otherwise
logical AND	& &	х & & у	1 if x and y are true,
			0 otherwise
array membership	in	i in a	1 if a [i] exists, 0 otherwise
matching	~ !~	\$1 ~ /x/	1 if the first field contains an x,
			0 otherwise
relational	< <= == !=	х == у	1 if x is equal to y,
	>= >		0 otherwise
concatenation		"a" "bc"	"abc"; there is no explicit
			concatenation operator
add, subtract	+ -	х + у	sum of x and y
multiply, divide, mod	* / %	х % у	remainder of x divided by y
unary plus and minus	+ -	-x	negated value of x
logical NOT	!	!\$1	1 if \$1 is zero or null,
			0 otherwise
exponentiation	^	х ^ у	х ^у
increment, decrement	++	++x, x++	add 1 to x
field	\$	\$i+1	value of i-th field, plus 1
grouping	()	\$(i++)	return i-th field, then increment i

TABLE A-7. EXPRESSION OPERATORS

Operators are listed in order of increasing precedence.

from C, except for the special form of for that iterates over arrays.

A single statement can always be replaced by a list of statements enclosed in braces. The statements in the list are separated by newlines or semicolons. Newlines may be inserted after any left brace and before any right brace.

The if-else statement has the form

```
if (expression)
    statement1
else
    statement2
```

The else *statement*₂ is optional. Newlines are optional after the right parenthesis, after *statement*₁, and after the keyword else. If else appears on the same line as *statement*₁, however, then a semicolon must terminate *statement*₁ if it is a single statement.

In an if-else statement, the test *expression* is evaluated first. If it is true, that is, either nonzero or nonnull, *statement*₁ is executed. If *expression* is false, that is, either zero or null, and else *statement*₂ is present, then *statement*₂ is executed.

Summary of Control-Flow Statements

{ statements }
statement grouping
if (expression) statement
if expression is true, execute statement
if (expression) statement ₁ else statement ₂
if <i>expression</i> is true, execute <i>statement</i> ₁ otherwise execute <i>statement</i> ₂
while (<i>expression</i>) statement
if expression is true, execute statement, then repeat
for (expression ₁ ; expression ₂ ; expression ₃) statement
equivalent to expression ₁ ; while (expression ₂) { statement; expression ₃ }
for (variable in array) statement
execute statement with variable set to each subscript in array in turn, in unspecified order
do <i>statement</i> while (<i>expression</i>)
execute statement; if expression is true, repeat
break
immediately leave innermost enclosing while, for, or do; illegal outside of loops
continue
start next iteration of innermost enclosing while, for, or do; illegal outside of loops
return expression
return from function with value expression if present.
next
start next iteration of main input loop; illegal inside function definition
nextfile
start next iteration of main input loop with the next input file; illegal inside function definition
exit
exit <i>expression</i>
go immediately to the END action; if within the END action, exit program entirely. Return <i>expression</i> as program status, or zero if there is no <i>expression</i> .

To eliminate any ambiguity, each else is associated with the closest previous unassociated if. For example, in the statement

if (e1) if (e2) s=1; else s=2

the else is associated with the second if. (The semicolon after s=1 is required, because the else appears on the same line.)

The while statement repeatedly executes a statement while a condition is true:

while (expression) statement

In this loop, *expression* is evaluated; if it is true, *statement* is executed and *expression* is tested again. The cycle repeats as long as *expression* is true, that is, until the *expression* becomes false. For example, this program prints all input fields, one per line:

```
{ i = 1
while (i <= NF) {
    print $i
    i++
  }
}</pre>
```

The loop stops when i reaches NF+1, and that is its value after the loop exits.

The for statement is a more general form of while:

for (expression₁; expression₂; expression₃)
 statement

The for statement has the same effect as

```
expression1
while (expression2) {
    statement
    expression3
}
```

so

does the same loop over the fields as the while example above. In the for statement, all three expressions are optional. If $expression_2$ is missing, the condition is taken to be always true, so for (;;) is an infinite loop.

An alternate version of the for statement that loops over array subscripts is described in Section A.2.5 below.

The do statement has the form

do *statement* while (*expression*)

Newlines are optional after the keyword do and after *statement*. If while appears on the same line as *statement*, however, then *statement* must be terminated by a semicolon if it is a single statement. The do loop executes *statement* once, then repeats *statement* as long as *expression* is true. It differs from the while and for in a critical way: its test for completion is at the bottom instead of the top, so it always goes through the loop at least once.

There are two statements for modifying how loops cycle: break and continue. The break statement causes an exit from the immediately enclosing while, for, or do. The continue statement causes the next iteration to begin; it causes execution to go to the test expression in the while and do, and to *expression*₃ in the for statement. Both break and continue are illegal outside of loops.

The return statement returns from a function, optionally with a value.

The next, nextfile, and exit statements control the outer loop that reads the input lines in an Awk program. The next statement causes Awk to fetch the next input line and begin matching patterns starting from the first pattern-action statement.

The nextfile statement causes Awk to close the current input file and begin processing the next input file if there is one.

In an END action, the exit statement causes the program to terminate immediately. In any other action, it causes the program to behave as if the end of the input had occurred; no more input is read, and the END actions, if any, are executed.

If an exit statement includes an expression:

exit expr

it causes Awk to return the value of *expr* as its exit status unless overridden by a subsequent error or exit. If there is no *expr*, the exit status is zero. In some operating systems, including Unix, the exit status may be tested by the program that invoked Awk.

A.2.4 Empty Statement

A semicolon by itself denotes the empty statement. In the following program, the body of the for loop is an empty statement.

The program prints all lines that contain an empty field.

A.2.5 Arrays

Awk provides one-dimensional arrays for storing strings and numbers. Arrays and array elements need not be declared, nor is there any need to specify how many elements an array has or will have. Like variables, array elements spring into existence by being mentioned; at birth, they have the numeric value 0 and the string value "".

As a simple example, the statement

x[NR] = \$0

assigns the current input line to element NR of the array x. In fact, it is often easiest to read the entire input into an array, then process it in any convenient order. For example, this variant of the program from Section 1.7 prints its input in reverse line order:

```
{ x[NR] = $0 }
END { for (i = NR; i > 0; i--) print x[i] }
```

The first action stores each input line in the array x, using the line number as a subscript; the real work is done in the END statement.

The characteristic that sets Awk arrays apart from those in most other languages is that subscripts are strings. This gives Awk a key-value capability like Python's dictionary data structure, or hash tables in Java or JavaScript, or maps in other languages. Arrays in Awk are called *associative arrays*, a terminology that predates dictionary and hash table.

The following program accumulates the populations of Asia and Africa in the array pop. The END action prints the total populations of these two continents.

On countries, this program generates

```
Asian population 3574 million
African population 320 million
```

Note that the subscripts are the string constants "Asia" and "Africa". If we had written pop[Asia] instead of pop["Asia"], the expression would have used the value of the variable Asia as the subscript, and because that variable is uninitialized, the values would have been accumulated in pop[""].

This example doesn't really need an associative array since there are only two elements, both named explicitly. Suppose instead that our task is to determine the total population for each continent. Associative arrays are ideally suited for this kind of aggregation. Any expression can be used as a subscript in an array reference, so

pop[\$4] += \$3

uses the string in the fourth field of the current input line to index the array pop and in that entry accumulates the value of the third field:

The subscripts of the array pop are the continent names; the values are the accumulated populations. This code works regardless of the number of continents; the output from the countries file is

```
Africa 320
South America 212
North America 459
Asia 3574
Europe 145
```

The last program used the for statement that loops over all subscripts of an array:

for (variable in array)
 statement

This loop executes *statement* with *variable* set in turn to each different subscript in the array. The order in which the subscripts are considered is implementation dependent. Results are unpredictable if elements are deleted or if new elements are added to the array by *statement*.

You can determine whether a particular subscript occurs in an array with the expression

subscript in A

This expression has the value 1 if A [subscript] already exists, and 0 otherwise. Thus, to test whether Africa is a subscript of the array pop you can say

if ("Africa" in pop) ...

This condition performs the test without the side effect of creating pop["Africa"], which would happen if you used

if (pop["Africa"] != "") ...

Note that neither is a test of whether the array pop contains an element with the value "Africa".

The delete Statement. An array element may be deleted with

```
delete array[subscript]
```

For example, this loop removes all the elements from the array pop:

for (i in pop)
 delete pop[i]

The statement

delete array

deletes the entire array, and thus delete pop is equivalent to the loop above.

The split Function. The function split(str, arr, fs) splits the string value of str into fields and stores them in the array arr; str is unchanged. The number of fields produced is returned as the value of split. The string value of the third argument, fs, determines the field separator. If there is no third argument, if -csv is set, splitting is done as for CSV; otherwise FS is used. The string fs may be a regular expression. The rules are as for input field splitting, which is discussed in Section A.5.1. The function call

split("7/4/76", arr, "/")

splits the string 7/4/76 into three fields using / as the separator; it stores 7 in arr["1"], 4 in arr["2"], and 76 in arr["3"].

If the source string is empty, the number of elements is always zero and the array is not set.

As a final special case, if the *fs* argument is the empty string "", the string *str* is split into individual characters, one character per array element.

Strings are versatile array subscripts, but the behavior of numeric subscripts as strings may sometimes appear counterintuitive. Since the string values of 1 and "1" are the same, arr[1] is the same as arr["1"]. But "01" is not the same string as "1" and the string "10" comes before the string "2".

Multidimensional Arrays. Awk does not support multidimensional arrays directly but it provides a simulation using one-dimensional arrays. If you write multidimensional subscripts like [i, j] or [s, p, q, r], Awk concatenates the components of the subscripts (with a separator between them) to synthesize a single subscript out of the multiple subscripts that you write. For example,

```
for (i = 1; i <= 10; i++)
  for (j = 1; j <= 10; j++)
      arr[i,j] = 0</pre>
```

creates an array of 100 elements whose subscripts appear to have the form 1, 1, 1, 2, and so

on. Internally, however, these subscripts are stored as strings of the form 1 SUBSEP 1, 1 SUBSEP 2, and so on. The built-in variable SUBSEP contains the value of the subscript-component separator; its default value is not a comma but the ASCII file separator character "034" or "x1C", a value that is unlikely to appear in normal text.

The test for array membership with multidimensional subscripts uses a parenthesized list of subscripts, such as

if ((i,j) in arr) ...

To loop over such an array, however, you would write

```
for (k in arr) ...
```

and use split (k, x, SUBSEP) if access to the individual subscript components is needed. Array elements cannot themselves be arrays.

A.3 User-Defined Functions

In addition to built-in functions, an Awk program can contain user-defined functions. Such a function is defined by a statement of the form

```
function name(parameter-list) {
    statements
}
```

A function definition can occur anywhere a pattern-action statement can. Thus, the general form of an Awk program is a sequence of pattern-action statements and function definitions separated by newlines or semicolons.

In a function definition, newlines are optional after the left brace and before the right brace of the function body. The parameter list is a sequence of zero or more variable names separated by commas; within the body of the function these variables refer to the arguments with which the function was called.

The body of a function definition may contain a return statement that returns control and perhaps a value to the caller:

return expression

The *expression* is optional, and so is the return statement itself, but the return value is "" and 0 if no *expression* is provided. If the last statement executed is not a return ("falling off the end of the function") the return value is also "" and 0.

For example, this function computes the maximum of its arguments:

```
function max(m, n) {
    return m > n ? m : n
}
```

The variables m and n are local to the function \max ; they are unrelated to any other variables of the same names elsewhere in the program.

A user-defined function can be used in any expression in any pattern-action statement or the body of any function definition. Each use is a *call* of the function. For example, the max function might be called like this:

```
{ print max($1, max($2, $3)) } # print maximum of $1, $2, $3
function max(m, n) {
   return m > n ? m : n
}
```

There cannot be any spaces between the function name and the left parenthesis of the argument list when the function is called.

If a user-defined function is called in the body of its own definition, that function is said to be *recursive*.

When a function is called with an argument like \$1, which is just an ordinary variable, the function is given a copy of the value of the variable, so the function manipulates the copy, not the variable itself. This means that the function cannot affect the value of the variable outside the function. (The jargon is that such variables, called "scalars," are passed "by value.") Arrays are not copied, however, so it is possible for the function to alter array elements or create new ones. (This is called passing "by reference.") The name of a function may not be used as a parameter, global array, or scalar.

To repeat, within a function definition, the parameters are local variables — they last only as long as the function is executing, and they are unrelated to variables of the same name elsewhere in the program. But *all other variables are global*; if a variable is not named in the parameter list, it is visible and accessible throughout the program.

This means that the only way to provide local variables for the private use of a function is to include them at the end of the parameter list in the function definition. Any variable in the parameter list for which no actual parameter is supplied in a call is a local variable, with null initial value. This is not a good design but it at least provides the necessary facility. We insert several spaces between the arguments and the local variables so they can be more easily distinguished. Omitting a local variable from this list is a common source of bugs.

A.4 Output

The print and printf statements generate output. The print statement is used for simple output; printf is used when careful formatting is required. Output from print and printf can be directed into files and pipes as well as to the terminal. These statements can be used in any mixture; the output comes out in the order in which it is generated.

A.4.1 The print Statement

The print statement has two equivalent forms:

```
print expr_1, expr_2, ..., expr_n
print (expr_1, expr_2, ..., expr_n)
```

Both forms print the string value of each expression separated by the output field separator followed by the output record separator. The statement

print

is an abbreviation for

Summary of Output Statements

```
print
   print $0 on standard output
print expression, expression, ...
   print expressions, separated by OFS, terminated by ORS
print expression, expression, ... > filename
   print on file filename instead of standard output
print expression, expression, ... >> filename
   append to file filename instead of overwriting previous contents
print expression, expression, ... | command
   print to standard input of command
printf(format, expression, expression, ...)
printf(format, expression, expression, ...) > filename
printf(format, expression, expression, ...) >> filename
printf(format, expression, expression, ...) | command
   printf statements are like print but the first argument specifies the output format
close (filename), close (command)
   break connection between print and filename or command
fflush (filename), fflush (command)
   force out any buffered output of filename or command
```

If an expression in the argument list of a print or printf statement contains a relational operator, either the expression or the argument list must be enclosed in parentheses. Pipes may not be available on non-Unix systems.

print \$0

To print a blank line, that is, a line with only a newline, use

print ""

The second form of the print statement encloses the argument list in parentheses, as in

print(\$1 ":", \$2)

Both forms of the print statement generate the same output but, as we will see, parentheses may be necessary for arguments containing relational operators.

A.4.2 Output Separators

The output field separator and output record separator are stored in the built-in variables OFS and ORS. Initially, OFS is set to a single space and ORS to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each line with a colon between the fields and two newlines after the second field:

```
BEGIN { OFS = ":"; ORS = "\n\n" }
{ print $1, $2 }
```

By contrast,

{ print \$1 \$2 }

prints the first and second fields with no intervening output field separator, because \$1 \$2 is a string consisting of the concatenation of the two fields.

A.4.3 The printf Statement

The printf statement generates formatted output. It is similar to that in C, though width qualifiers like h and l have no effect.

printf(format, $expr_1$, $expr_2$, ..., $expr_n$)

The *format* argument is always required; it is an expression whose string value contains both literal text to be printed and specifications of how the expressions in the argument list are to be formatted, as in Table A-8. Each specification begins with %, ends with a character that determines the conversion, and may include modifiers:

_	left-justify expression in its field
+	always print a sign
0	pad with zeros instead of spaces
width	pad result to this width as needed; leading 0 pads with zeros
.prec	maximum string width, or digits to right of decimal point

If a * appears in a specification, it is replaced by the numeric value of the next argument, so widths and precisions can be provided dynamically.

CHARACTER	PRINT EXPRESSION AS
С	single UTF-8 character (code point)
d or i	decimal integer
e or E	[-]d.dddddde[+-]dd or [-]d.ddddddE[+-]dd
f	[-]ddd.ddddd
g or G	e or f conversion, whichever is shorter, with nonsignificant zeros suppressed
0	unsigned octal number
u	unsigned integer
S	string
x or X	unsigned hexadecimal number
00	print a %; no argument is consumed

TABLE A-8. PRINTF FORMAT-CONTROL CHARACTERS

Table A-9 contains some examples of specifications, data, and the corresponding output. Output produced by printf does not contain any newlines unless you put them in explicitly.

A.4.4 Output into Files

The redirection operators > and >> are used to put output into files instead of the standard output. The following program will put the first and third fields of all input lines into two files: bigpop if the third field is greater than 1000, and smallpop otherwise:

\$3 > 1000 { print \$1, \$3 >"bigpop" } \$3 <= 1000 { print \$1, \$3 >"smallpop" }

fmt	\$1	<pre>printf(fmt, \$1)</pre>
%C	97	a
%d	97.5	97
%5d	97.5	97
°е	97.5	9.750000e+01
%f	97.5	97.500000
%7.2f	97.5	97.50
%g	97.5	97.5
%.6g	97.5	97.5
80	97	141
8060	97	000141
°X	97	61
⁸ S	January	January
%10s	January	January
%-10s	January	January
%.3s	January	Jan
%10.3s	January	Jan
%-10.3s	January	Jan
90 90 90	January	00

TABLE A-9. EXAMPLES OF PRINTF SPECIFICATIONS

Notice that the filenames have to be quoted; without quotes, bigpop and smallpop are merely uninitialized variables. Filenames can be variables or expressions as well:

{ print(\$1, \$3) > (\$3 > 1000 ? "bigpop" : "smallpop") }

does the same job, and the program

{ print > \$1 }

puts each input line into a file named by its first field.

In print and printf statements, if an expression in the argument list contains a relational operator, then either that expression or the argument list needs to be parenthesized. This rule eliminates any potential ambiguity arising from the redirection operator >. In

{ print \$1, \$2 > \$3 }

> is the redirection operator, and hence not part of the second expression, so the values of the first two fields are written to the file named in the third field. If you want the second expression to include the > operator, use parentheses:

```
{ print $1, ($2 > $3) }
```

It is also important to note that a redirection operator opens a file only once; each successive print or printf statement adds more data to the open file. When the redirection operator > is used, the file is initially cleared before any output is written to it. If >> is used instead of >, the file is not cleared when opened; output is appended after the original contents of the file.

There are three special filenames for pre-defined input and output streams: "/dev/stdin", "/dev/stdout", and "/dev/stderr" represent the standard input, standard output, and standard error streams of the program. The name "-" may also be used for the standard input.

A.4.5 Output into Pipes

It is also possible to direct output into a pipe instead of a file on systems that support pipes. The statement

```
print | command
```

causes the output of print to be piped into the *command*.

Suppose we want to create a list of continent-population pairs, sorted in reverse numeric order by population. The program below accumulates in an array pop the population values in the third field for each of the distinct continent names in the fourth field. The END action prints each continent name and its population, and pipes this output into a suitable sort command.

```
# print continents and populations, sorted by population
BEGIN { FS = "\t" }
        { pop[$4] += $3 }
END { for (c in pop)
            printf("%15s\t%6d\n", c, pop[c]) | "sort -t'\t' -k2 -rn"
        }
```

This yields

	Asia	3574
North	America	459
	Africa	320
South	America	212
	Europe	145

Another use for a pipe is writing onto the standard error file on Unix systems; output written there appears on the user's terminal instead of the standard output. There are several older idioms for writing on the standard error:

```
print message | "cat 1>&2"  # redirect cat output to stderr
system("echo '" message "' 1>&2")  # redirect echo output to stderr
print message > "/dev/tty"  # write directly on terminal
```

but the easiest idiom with newer versions of Awk is instead to write to /dev/stderr.

Although most of our examples show literal strings enclosed in quotes, command lines and filenames can be specified by any expression. In print statements involving redirection of output, the files or pipes are identified by their names; that is, the pipe in the program above is literally named

sort -t'\t' -k2 -rn

Normally, a file or pipe is created and opened only once during the run of a program. If the file or pipe is explicitly closed and then reused, it will be reopened.
A.4.6 Closing Files and Pipes

The statement close (*expr*) closes a file or pipe denoted by *expr*; the string value of *expr* must be exactly the same as the string used to create the file or pipe in the first place. Thus

close("sort -t'\t' -k2 -rn")

closes the sort pipe opened above.

close is necessary if you intend to write a file, then read it later in the same program. There are also system-defined limits on the number of files and pipes that can be open at the same time.

close is a function; it returns the value returned by the underlying fclose function or exit status for a pipeline.

The fflush function forces out any output that has been collected for a file or pipe; fflush() or fflush("") flush all output files and pipes.

A.5 Input

There are several ways of providing input to an Awk program. It's obviously possible to just type input at the keyboard, but the most common arrangement is to put input data in a file, say data, and then type

awk 'program' data

Awk reads its standard input if no filenames are given; thus, a second common arrangement is to have another program pipe its output into Awk. For example, the program grep selects input lines containing a specified regular expression, and is part of many Unix programmers' muscle memory. They might instinctively type

grep Asia countries | awk 'program'

to use grep to find the lines containing Asia and pass them on to Awk for subsequent processing.

Use "-" or /dev/stdin on the command line to read the standard input in the middle of a list of files.

Note that literal escaped characters like $\n \text{ or } \007$ are not interpreted nor are they in any way special when they appear in an input stream; they are just literal byte sequences. The only interpretation on input is that apparently-numeric values like scientific notation and explicitly signed instances of nan and inf will be stored with a numeric value as well as a string value.

A.5.1 Input Separators

The default value of the built-in variable FS is " ", that is, a single space. When FS has this specific value, input fields are separated by spaces and/or tabs, and leading spaces and tabs are discarded, so each of the following lines has the same first field:

```
field1
field1
field1 field2
```

When FS has any other value, leading spaces and tabs are not discarded.

The field separator can be changed by assigning a string to the built-in variable FS. If the string is longer than one character, it is taken to be a regular expression. The leftmost longest nonnull and nonoverlapping substrings matched by that regular expression become the field separators in the current input line. For example,

BEGIN { FS = "[t]+" }

makes every string consisting of spaces and tabs into a field separator.

When FS is set to a single character other than space, that literal character becomes the field separator. This convention makes it easy to use regular expression metacharacters as field separators:

FS = "|"

makes | a field separator. But note that something indirect like

FS = "[]"

is required to set the field separator to a single space.

FS can also be set on the command line with the -F argument. The command line

awk -F'[\t]+' 'program'

sets the field separator to the same strings as the BEGIN action shown above.

Finally, if the -csv argument is used, fields are treated as comma-separated values, and the value of FS is irrelevant.

A.5.2 CSV Input

Comma-separated values, or CSV, is a widely used format for spreadsheet data. As we said earlier, CSV is not rigorously defined, but generally 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 records are terminated by an unquoted newline, perhaps preceded by a carriage return (\r) for files that originated on Windows. Input fields in CSV files may contain embedded newline characters. A quoted $\r\n$ is converted to \n . A quoted $\r \n$ is left alone.

A.5.3 Multiline Records

By default, records are separated by newlines, so the terms "line" and "record" are normally synonymous. The default record separator can be changed, however, by assigning a new value to the built-in record-separator variable RS.

If RS is set to the null string, as in

BEGIN { RS = "" }

then records are separated by one or more blank lines and each record can therefore occupy several lines. Setting RS back to newline with the assignment $RS = "\n"$ restores the default behavior. With multiline records, no matter what value FS has, newline is always one of the field separators. Input fields may not contain newlines unless the --csv option has been used.

A common way to process multiline records is to use

BEGIN { RS = ""; FS = "\n" }

to set the record separator to one or more blank lines and the field separator to a newline alone; each line is thus a separate field. Section 4.4 contains more discussion of how to handle multiline records.

The RS variable can be a regular expression, so it's possible to separate records by text strings more complicated than just a single character. For example, in a well-structured HTML document, individual paragraphs might be separated by tags. By setting RS to <[Pp]>, an input file could be split into records that were each one HTML paragraph.

A.5.4 The getline Function

The function getline reads input either from the current input or from a file or pipe. By itself, getline fetches the next input record, performs the normal field-splitting operations on it, and sets NF, NR, and FNR. It returns 1 if there was a record present, 0 if end-of-file was encountered, and -1 if some error occurred (such as failure to open a file).

The expression getline x reads the next record into the variable x and increments NR and FNR. No splitting is done; NF is not set.

The expression

```
getline <"file"
```

reads from file instead of the current input. It has no effect on NR or FNR, but field splitting is performed and NF is set.

The expression

getline x <"file"</pre>

gets the next record from file into the variable x; no splitting is done, and NF, NR, and FNR are untouched.

If the filename is "-", the standard input is read; the filename "/dev/stdin" is equivalent.

Table A-10 summarizes the forms of the getline function. The value of each expression is the value returned by getline.

EXPRESSION	Sets
getline	\$0, NF, NR, FNR
getline var	<i>var</i> , NR, FNR
getline < <i>file</i>	\$0,NF
getline var <file< td=""><td>var</td></file<>	var
cmd getline	\$0,NF
cmd getline var	var

TABLE A-10. GETLINE FUNCTION

As an example, this program copies its input to its output, except that each line like

#include "filename"

is replaced by the contents of the file filename.

```
# include - replace #include "f" by contents of file f
/^#include/ {
    gsub(/"/, "", $2)
    while (getline x <$2 > 0)
        print x
        close(x)
        next
}
{ print }
```

It is also possible to pipe the output of another command directly into getline. For example, the statement

executes the Unix program who (once only) and pipes its output into getline. The output of who is a list of the users logged in. Each iteration of the while loop reads one more line from this list and increments the variable n, so after the while loop terminates, n contains a count of the number of users. Similarly, the expression

"date" | getline d

pipes the output of the date command into the variable d, thus setting d to the current date. Again, input pipes may not be available on non-Unix systems.

In all cases involving getline, you should be aware of the possibility of an error return if the file can't be accessed. Although it's appealing to write

while (getline <"file") ... # Dangerous</pre>

that's an infinite loop if file doesn't exist, because with a nonexistent file getline returns -1, a nonzero value that is interpreted as true. The preferred way is

while (getline <"file" > 0) ... # Safe

Here the loop will be executed only when getline returns 1, which it does for each input line it reads.

A.5.5 Command-Line Arguments and Variable Assignments

As we have seen, an Awk command line can have several forms:

awk 'program' f1 f2 ... awk -f progfile f1 f2 ... awk -Fsep 'program' f1 f2 ... awk -Fsep -f progfile f1 f2 ... awk --csv f1 f2 ... awk -v var=value f1 f2 ... awk --version

In these command lines, f1, f2, etc., are command-line arguments that normally represent filenames; the name "-" may be used for the standard input. The argument --csv enables CSV input processing.

The special argument -- can be used to end the list of options.

If a filename has the form *var=value*, however, it is treated as an assignment of *value* to the Awk variable *var*, performed when that argument would otherwise be accessed as a file. This type of assignment allows variables to be changed before and after a file is read.

The command-line arguments are available to the Awk program in a built-in array called ARGV. The value of the built-in variable ARGC is one more than the number of arguments. With the command line

```
awk -f progfile a v=1 b
```

ARGC has the value 4, ARGV[0] contains awk, ARGV[1] contains a, ARGV[2] contains v=1, and ARGV[3] contains b. ARGC is one more than the number of arguments because awk, the name of the command, is counted as argument zero, as it is in C programs. If the Awk program appears on the command line, however, the program is not treated as an argument, nor is -f filename or any -F option. For example, with the command line

awk $-F' \setminus t'$ '\$3 > 100' countries

ARGC is 2, ARGV[0] is awk and ARGV[1] is countries.

The following program echoes its command-line arguments (and a spurious space):

```
# echo - print command-line arguments
BEGIN {
   for (i = 1; i < ARGC; i++)
        printf "%s ", ARGV[i]
        printf "\n"
}</pre>
```

Notice that everything happens in the BEGIN action: because there are no other pattern-action statements, the arguments are never treated as filenames, and no input is read.

Another program using command-line arguments is seq, which generates sequences of integers:

```
# seq - print sequences of integers
#
  input: arguments q, p q, or p q r; q \ge p; r \ge 0
#
   output: integers 1 to q, p to q, or p to q in steps of r
BEGIN {
    if (ARGC == 2)
        for (i = 1; i <= ARGV[1]; i++)</pre>
            print i
    else if (ARGC == 3)
        for (i = ARGV[1]; i <= ARGV[2]; i++)</pre>
            print i
    else if (ARGC == 4)
        for (i = ARGV[1]; i <= ARGV[2]; i += ARGV[3])
            print i
}
```

The commands

awk -f seq 10 awk -f seq 1 10 awk -f seq 1 10 1 all generate the integers one through ten.

The arguments in ARGV may be modified or added to, and ARGC may be altered. As each input file ends, Awk treats the next nonnull element of ARGV (up through the current value of ARGC-1) as the name of the next input file. Thus setting an element of ARGV to null means that it will not be treated as an input file.

Increasing ARGC and adding elements to ARGV cause more filenames to be processed.

A.6 Interaction with Other Programs

This section describes some of the ways in which Awk programs can cooperate with other commands. The discussion applies primarily to the Unix operating system; the examples here may fail or work differently on non-Unix systems.

A.6.1 The system Function

The built-in function system (*expression*) executes the command given by the string value of *expression*. The value returned by system is the status returned by the command that was executed, as with close.

For example, we can build another version of the file-inclusion program of Section A.5.4 like this:

```
$1 == "#include" {
   gsub(/"/, "", $2)
   system("cat " $2)
   next
}
{ print }
```

If the first field is #include, quotes are removed, and the Unix command cat is called to print the file named in the second field. Other lines are just copied.

A.6.2 Making a Shell Command from an Awk Program

In all of the examples so far, the Awk program was in a file and fetched with the -f flag, or it appeared on the command line enclosed in single quotes, like this:

awk '{ print \$1 }' ...

Since Awk uses many of the same characters as the shell does, such as \$ and ", surrounding the program with single quotes ensures that the shell will pass the entire program unchanged to Awk.

Both methods of invoking the Awk program require some typing. To reduce the number of keystrokes, we might want to put both the command and the program into an executable file, and invoke the command by typing just the name of the file.

Suppose we want to create a command field1 that will print the first field of each line of input. This is easy: we put

```
awk '{print $1}' $*
```

into the file field1, and make the file executable by typing the Unix command

\$ chmod +x field1

We can now print the first field of each line of a set of files by typing

field1 filenames ...

Now, consider writing a more general command field that will print an arbitrary combination of fields from each line of its input; in other words, the command

field $n_1 n_2 \ldots$ file₁ file₂ ...

will print the specified fields in the specified order. How do we get the value of each n_i into the Awk program each time it is run and how do we distinguish the n_i 's from the filename arguments?

There are several ways to do this if one is adept in shell programming. The simplest way that uses only Awk, however, is to scan through the built-in array ARGV to process the n_i 's, resetting each such argument to the null string so that it is not treated as a filename.

```
# field - print named fields of each input line
    usage: field n n n ... file file file ...
#
awk '
BEGIN {
    for (i = 1; ARGV[i] ~ /^[0-9]+$/; i++) { # collect numbers
        fld[++nf] = ARGV[i]
        ARGV[i] = ""
    }
    if (i \ge ARGC)
                    # no file names so force stdin
        ARGV[ARGC++] = "-"
}
{
    for (i = 1; i <= nf; i++)</pre>
        printf("%s%s", $fld[i], i < nf ? " " : "\n")</pre>
}
' $*
```

This version can deal with either standard input or a list of filename arguments, and with any number of fields in any order.

A.7 Summary

As we said earlier, this is a long manual, packed with details, and you are dedicated indeed if you have read every word to get here. You will find that it pays to go back and reread sections from time to time, either to see precisely how something works, or because one of the examples suggests a construction that you might not have tried before.

Awk, like any language, is best learned by experience and practice, so we encourage you to write your own programs. They don't have to be big or complicated — you can usually learn how some feature works or test some crucial point with only a couple of lines of code, and you can just type in data to see how the program behaves.

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 acc 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

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

fflush 202

heapsort, profiling 143-144 Herbst, R. T. xiii hexadecimal escape, \x 173 Hill Farmstead 49 histogram program 53 Hoare, C. A. R. 135 Hovt, 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

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 1s 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 - 182match, 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

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 floatingpoint 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

n log n algorithm 137, 140

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 1 201 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 charfreg 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 pl2check 58 percent 53 prep1 68 prep2 70

printprof 142 gawk 82 quiz 92 quote 31 randline 86 rtsort 148 sentgen 89 seg 206 sortgen 114 sumcomma 54 table 72 test framework 135 tsort 146 unbundle 60 word count 13, 92 wordfreg 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 gawk 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 2 173 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

This page intentionally left blank



Register Your Product at informit.com/register Access additional benefits and save up to 65%* on your next purchase

- Automatically receive a coupon for 35% off books, eBooks, and web editions and 65% off video courses, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.**
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's leading learning company. At informit.com, you can

- Shop our books, eBooks, and video training. Most eBooks are DRM-Free and include PDF and EPUB files.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletter (informit.com/newsletters).
- Access thousands of free chapters and video lessons.
- Enjoy free ground shipping on U.S. orders.*

* Offers subject to change.

** Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

Connect with InformIT—Visit informit.com/community



y twitter.com/informit

