Effective AWK Programming

A User's Guide for GNU Awk
Edition 1.0.6
July, 2000

Arnold D. Robbins

"To boldly go where no man has gone before" is a Registered Trademark of Paramount Pictures Corporation.

Copyright © 1989, 1991, 1992, 1993, 1996-2000 Free Software Foundation, Inc.

This is Edition 1.0.6 of *Effective AWK Programming*,
for the 3.0.6 (or later) version of the GNU implementation of AWK.

Published by:
Free Software Foundation
59 Temple Place — Suite 330
Boston, MA 02111-1307 USA
Phone: +1-617-542-5942
Fax: +1-617-542-2652
Email: gnu@gnu.org
URL: http://www.gnu.org/

ISBN 1-882114-26-4

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

Cover art by Etienne Suvasa.

To Miriam, for making me complete.

To Chana, for the joy you bring us.

To Rivka, for the exponential increase.

To Nachum, for the added dimension.

To Malka, for the new beginning.

Preface

This book teaches you about the `awk` language and how you can use it effectively. You should already be familiar with basic system commands, such as `cat` and `ls`,[1] and basic shell facilities, such as Input/Output (I/O) redirection and pipes.

Implementations of the `awk` language are available for many different computing environments. This book, while describing the `awk` language in general, also describes a particular implementation of `awk` called `gawk` (which stands for "GNU Awk"). `gawk` runs on a broad range of Unix systems, ranging from 80386 PC-based computers, up through large scale systems, such as Crays. `gawk` has also been ported to MS-DOS and OS/2 PC's, Atari and Amiga micro-computers, and VMS.

History of `awk` and `gawk`

The name `awk` comes from the initials of its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. The original version of `awk` was written in 1977 at AT&T Bell Laboratories. In 1985 a new version made the programming language more powerful, introducing user-defined functions, multiple input streams, and computed regular expressions. This new version became generally available with Unix System V Release 3.1. The version in System V Release 4 added some new features and also cleaned up the behavior in some of the "dark corners" of the language. The specification for `awk` in the POSIX Command Language and Utilities standard further clarified the language based on feedback from both the `gawk` designers, and the original Bell Labs `awk` designers.

The GNU implementation, `gawk`, was written in 1986 by Paul Rubin and Jay Fenlason, with advice from Richard Stallman. John Woods contributed parts of the code as well. In 1988 and 1989, David Trueman, with help from Arnold Robbins, thoroughly reworked `gawk` for compatibility with the newer `awk`. Current development focuses on bug fixes, performance improvements, standards compliance, and occasionally, new features.

The GNU Project and This Book

The Free Software Foundation (FSF) is a non-profit organization dedicated to the production and distribution of freely distributable software. It was founded by Richard M. Stallman, the author of the original Emacs editor. GNU Emacs is the most widely used version of Emacs today.

---

[1] These commands are available on POSIX compliant systems, as well as on traditional Unix based systems. If you are using some other operating system, you still need to be familiar with the ideas of I/O redirection and pipes.

The GNU project is an on-going effort on the part of the Free Software Foundation to create a complete, freely distributable, POSIX compliant computing environment. (GNU stands for "GNU's not Unix".) The FSF uses the "GNU General Public License" (or GPL) to ensure that source code for their software is always available to the end user. A copy of the GPL is included for your reference (see [GNU GENERAL PUBLIC LICENSE], page 303). The GPL applies to the C language source code for `gawk`.

A shell, an editor (Emacs), highly portable optimizing C, C++, and Objective-C compilers, a symbolic debugger, and dozens of large and small utilities (such as `gawk`), have all been completed and are freely available. As of this writing (early 1997), the GNU operating system kernel (the HURD), has been released, but is still in an early stage of development.

Until the GNU operating system is more fully developed, you should consider using Linux, a freely distributable, Unix-like operating system for 80386, DEC Alpha, Sun SPARC and other systems. There are many books on Linux. One freely available one is *Linux Installation and Getting Started*, by Matt Welsh. Many Linux distributions are available, often in computer stores or bundled on CD-ROM with books about Linux. (There are three other freely available, Unix-like operating systems for 80386 and other systems, NetBSD, FreeBSD,and OpenBSD. All are based on the 4.4-Lite Berkeley Software Distribution, and they use recent versions of `gawk` for their versions of `awk`.)

This book you are reading now is actually free. The information in it is freely available to anyone, the machine readable source code for the book comes with `gawk`, and anyone may take this book to a copying machine and make as many copies of it as they like. (Take a moment to check the copying permissions on the Copyright page.)

If you paid money for this book, what you actually paid for was the book's nice printing and binding, and the publisher's associated costs to produce it. We have made an effort to keep these costs reasonable; most people would prefer a bound book to over 330 pages of photo-copied text that would then have to be held in a loose-leaf binder (not to mention the time and labor involved in doing the copying). The same is true of producing this book from the machine readable source; the retail price is only slightly more than the cost per page of printing it on a laser printer.

This book itself has gone through several previous, preliminary editions. I started working on a preliminary draft of *The GAWK Manual*, by Diane Close, Paul Rubin, and Richard Stallman in the fall of 1988. It was around 90 pages long, and barely described the original, "old" version of `awk`. After substantial revision, the first version of the *The GAWK Manual* to be released was Edition 0.11 Beta in October of 1989. The manual then underwent more substantial revision for Edition 0.13 of December 1991. David Trueman, Pat Rankin, and Michal Jaegermann contributed sections of the manual for Edition 0.13. That edition was published by the FSF as a bound book early in 1992. Since then there have been several minor revisions,

notably Edition 0.14 of November 1992 that was published by the FSF in January of 1993, and Edition 0.16 of August 1993.

Edition 1.0 of *Effective AWK Programming* represents a significant reworking of *The GAWK Manual*, with much additional material. The FSF and I agree that I am now the primary author. I also felt that it needed a more descriptive title.

Effective AWK Programming will undoubtedly continue to evolve. An electronic version comes with the `gawk` distribution from the FSF. If you find an error in this book, please report it! See Section B.7 [Reporting Problems and Bugs], page 285, for information on submitting problem reports electronically, or write to me in care of the FSF.

Acknowledgements

I would like to acknowledge Richard M. Stallman, for his vision of a better world, and for his courage in founding the FSF and starting the GNU project.

The initial draft of *The GAWK Manual* had the following acknowledgements:

> Many people need to be thanked for their assistance in producing this manual. Jay Fenlason contributed many ideas and sample programs. Richard Mlynarik and Robert Chassell gave helpful comments on drafts of this manual. The paper *A Supplemental Document for* `awk` by John W. Pierce of the Chemistry Department at UC San Diego, pinpointed several issues relevant both to `awk` implementation and to this manual, that would otherwise have escaped us.

The following people provided many helpful comments on Edition 0.13 of *The GAWK Manual*: Rick Adams, Michael Brennan, Rich Burridge, Diane Close, Christopher ("Topher") Eliot, Michael Lijewski, Pat Rankin, Miriam Robbins, and Michal Jaegermann.

The following people provided many helpful comments for Edition 1.0 of *Effective AWK Programming*: Karl Berry, Michael Brennan, Darrel Hankerson, Michal Jaegermann, Michael Lijewski, and Miriam Robbins. Pat Rankin, Michal Jaegermann, Darrel Hankerson and Scott Deifik updated their respective sections for Edition 1.0.

Robert J. Chassell provided much valuable advice on the use of Texinfo. He also deserves special thanks for convincing me *not* to title this book *How To Gawk Politely*. Karl Berry helped significantly with the TEX part of Texinfo.

David Trueman deserves special credit; he has done a yeoman job of evolving `gawk` so that it performs well, and without bugs. Although he is no longer involved with `gawk`, working with him on this project was a significant pleasure.

Scott Deifik, Darrel Hankerson, Kai Uwe Rommel, Pat Rankin, and Michal Jaegermann (in no particular order) are long time members of the `gawk` "crack portability team." Without their hard work and help, `gawk` would not be nearly the fine program it is today. It has been and continues to be a pleasure working with this team of fine people.

Jeffrey Friedl provided invaluable help in tracking down a number of last minute problems with regular expressions in `gawk` 3.0.

David and I would like to thank Brian Kernighan of Bell Labs for invaluable assistance during the testing and debugging of `gawk`, and for help in clarifying numerous points about the language. We could not have done nearly as good a job on either `gawk` or its documentation without his help.

I would like to thank Marshall and Elaine Hartholz of Seattle, and Dr. Bert and Rita Schreiber of Detroit for large amounts of quiet vacation time in their homes, which allowed me to make significant progress on this book and on `gawk` itself. Phil Hughes of SSC contributed in a very important way by loaning me his laptop Linux system, not once, but twice, allowing me to do a lot of work while away from home.

Finally, I must thank my wonderful wife, Miriam, for her patience through the many versions of this project, for her proof-reading, and for sharing me with the computer. I would like to thank my parents for their love, and for the grace with which they raised and educated me. I also must acknowledge my gratitude to G-d, for the many opportunities He has sent my way, as well as for the gifts He has given me with which to take advantage of those opportunities.

Arnold Robbins
Atlanta, Georgia
February, 1997

1 Introduction

If you are like many computer users, you would frequently like to make changes in various text files wherever certain patterns appear, or extract data from parts of certain lines while discarding the rest. To write a program to do this in a language such as C or Pascal is a time-consuming inconvenience that may take many lines of code. The job may be easier with `awk`.

The `awk` utility interprets a special-purpose programming language that makes it possible to handle simple data-reformatting jobs with just a few lines of code.

The GNU implementation of `awk` is called `gawk`; it is fully upward compatible with the System V Release 4 version of `awk`. `gawk` is also upward compatible with the POSIX specification of the `awk` language. This means that all properly written `awk` programs should work with `gawk`. Thus, we usually don't distinguish between `gawk` and other `awk` implementations.

Using `awk` you can:

- manage small, personal databases
- generate reports
- validate data
- produce indexes, and perform other document preparation tasks
- even experiment with algorithms that can be adapted later to other computer languages

1.1 Using This Book

The term `awk` refers to a particular program, and to the language you use to tell this program what to do. When we need to be careful, we call the program "the `awk` utility" and the language "the `awk` language." The term `gawk` refers to a version of `awk` developed as part the GNU project. The purpose of this book is to explain both the `awk` language and how to run the `awk` utility.

The main purpose of the book is to explain the features of `awk`, as defined in the POSIX standard. It does so in the context of one particular implementation, `gawk`. While doing so, it will also attempt to describe important differences between `gawk` and other `awk` implementations. Finally, any `gawk` features that are not in the POSIX standard for `awk` will be noted.

This book has the difficult task of being both tutorial and reference. If you are a novice, feel free to skip over details that seem too complex. You should also ignore the many cross references; they are for the expert user, and for the on-line Info version of the document.

The term `awk` *program* refers to a program written by you in the `awk` programming language.

See Chapter 2 [Getting Started with `awk`], page 9, for the bare essentials you need to know to start using `awk`.

Some useful "one-liners" are included to give you a feel for the `awk` language (see Chapter 3 [Useful One Line Programs], page 21).

Many sample `awk` programs have been provided for you (see Chapter 15 [A Library of `awk` Functions], page 167; also see Chapter 16 [Practical `awk` Programs], page 201).

The entire `awk` language is summarized for quick reference in Appendix A [`gawk` Summary], page 253. Look there if you just need to refresh your memory about a particular feature.

If you find terms that you aren't familiar with, try looking them up in the glossary (see Appendix D [Glossary], page 295).

Most of the time complete `awk` programs are used as examples, but in some of the more advanced sections, only the part of the `awk` program that illustrates the concept being described is shown.

While this book is aimed principally at people who have not been exposed to `awk`, there is a lot of information here that even the `awk` expert should find useful. In particular, the description of POSIX `awk`, and the example programs in Chapter 15 [A Library of `awk` Functions], page 167, and Chapter 16 [Practical `awk` Programs], page 201, should be of interest.

Dark Corners

> *Who opened that window shade?!?*
> Count Dracula

Until the POSIX standard (and *The Gawk Manual*), many features of `awk` were either poorly documented, or not documented at all. Descriptions of such features (often called "dark corners") are noted in this book with "(d.c.)". They also appear in the index under the heading "dark corner."

1.2 Typographical Conventions

This book is written using Texinfo, the GNU documentation formatting language. A single Texinfo source file is used to produce both the printed and on-line versions of the documentation. Because of this, the typographical conventions are slightly different than in other books you may have read.

Examples you would type at the command line are preceded by the common shell primary and secondary prompts, '`$`' and '`>`'. Output from the command is preceded by the glyph "⊣". This typically represents the command's standard output. Error messages, and other output on the command's standard error, are preceded by the glyph "error ". For example:

```
$ echo hi on stdout
⊣ hi on stdout
$ echo hello on stderr 1>&2
error  hello on stderr
```

In the text, command names appear in `this font`, while code segments appear in the same font and quoted, '`like this`'. Some things will be emphasized *like this*, and if a point needs to be made strongly, it will be done **like this**. The first occurrence of a new term is usually its *definition*, and appears in the same font as the previous occurrence of "definition" in this sentence. File names are indicated like this: '`/path/to/ourfile`'.

Characters that you type at the keyboard look ***`like this`***. In particular, there are special characters called "control characters." These are characters that you type by holding down both the ***`CONTROL`*** key and another key, at the same time. For example, a ***`Control-d`*** is typed by first pressing and holding the ***`CONTROL`*** key, next pressing the ***`d`*** key, and finally releasing both keys.

1.3 Data Files for the Examples

Many of the examples in this book take their input from two sample data files. The first, called '`BBS-list`', represents a list of computer bulletin board systems together with information about those systems. The second data file, called '`inventory-shipped`', contains information about shipments on a monthly basis. In both files, each line is considered to be one *record*.

In the file '`BBS-list`', each record contains the name of a computer bulletin board, its phone number, the board's baud rate(s), and a code for the number of hours it is operational. An '`A`' in the last column means the board operates 24 hours a day. A '`B`' in the last column means the board operates evening and weekend hours, only. A '`C`' means the board operates only on weekends.

```
aardvark     555-5553     1200/300          B
alpo-net     555-3412     2400/1200/300     A
barfly       555-7685     1200/300          A
bites        555-1675     2400/1200/300     A
camelot      555-0542     300               C
core         555-2912     1200/300          C
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sdace        555-3430     2400/1200/300     A
sabafoo      555-2127     1200/300          C
```

The second data file, called '`inventory-shipped`', represents information about shipments during the year. Each record contains the month of the year, the number of green crates shipped, the number of red boxes

shipped, the number of orange bags shipped, and the number of blue packages shipped, respectively. There are 16 entries, covering the 12 months of one year and four months of the next year.

```
Jan  13  25  15 115
Feb  15  32  24 226
Mar  15  24  34 228
Apr  31  52  63 420
May  16  34  29 208
Jun  31  42  75 492
Jul  24  34  67 436
Aug  15  34  47 316
Sep  13  55  37 277
Oct  29  54  68 525
Nov  20  87  82 577
Dec  17  35  61 401

Jan  21  36  64 620
Feb  26  58  80 652
Mar  24  75  70 495
Apr  21  70  74 514
```

2 Getting Started with awk

The basic function of awk is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, awk performs specified actions on that line. awk keeps processing input lines in this way until the end of the input files are reached.

Programs in awk are different from programs in most other languages, because awk programs are *data-driven*; that is, you describe the data you wish to work with, and then what to do when you find it. Most other languages are *procedural*; you have to describe, in great detail, every step the program is to take. When working with procedural languages, it is usually much harder to clearly describe the data your program will process. For this reason, awk programs are often refreshingly easy to both write and read.

When you run awk, you specify an awk *program* that tells awk what to do. The program consists of a series of *rules*. (It may also contain *function definitions*, an advanced feature which we will ignore for now. See Chapter 13 [User-defined Functions], page 151.) Each rule specifies one pattern to search for, and one action to perform when that pattern is found.

Syntactically, a rule consists of a pattern followed by an action. The action is enclosed in curly braces to separate it from the pattern. Rules are usually separated by newlines. Therefore, an awk program looks like this:

```
pattern { action }
pattern { action }
...
```

2.1 A Rose By Any Other Name

The awk language has evolved over the years. Full details are provided in Chapter 17 [The Evolution of the awk Language], page 247. The language described in this book is often referred to as "new awk."

Because of this, many systems have multiple versions of awk. Some systems have an awk utility that implements the original version of the awk language, and a nawk utility for the new version. Others have an oawk for the "old awk" language, and plain awk for the new one. Still others only have one version, usually the new one.[1]

All in all, this makes it difficult for you to know which version of awk you should run when writing your programs. The best advice we can give here is to check your local documentation. Look for awk, oawk, and nawk, as well as for gawk. Chances are, you will have some version of new awk on your system, and that is what you should use when running your programs. (Of course, if you're reading this book, chances are good that you have gawk!)

[1] Often, these systems use gawk for their awk implementation!

Throughout this book, whenever we refer to a language feature that should be available in any complete implementation of POSIX `awk`, we simply use the term `awk`. When referring to a feature that is specific to the GNU implementation, we use the term `gawk`.

2.2 How to Run `awk` Programs

There are several ways to run an `awk` program. If the program is short, it is easiest to include it in the command that runs `awk`, like this:

```
awk 'program' input-file1 input-file2 ...
```

where *program* consists of a series of patterns and actions, as described earlier. (The reason for the single quotes is described below, in Section 2.2.1 [One-shot Throw-away `awk` Programs], page 10.)

When the program is long, it is usually more convenient to put it in a file and run it with a command like this:

```
awk -f program-file input-file1 input-file2 ...
```

2.2.1 One-shot Throw-away `awk` Programs

Once you are familiar with `awk`, you will often type in simple programs the moment you want to use them. Then you can write the program as the first argument of the `awk` command, like this:

```
awk 'program' input-file1 input-file2 ...
```

where *program* consists of a series of *patterns* and *actions*, as described earlier.

This command format instructs the *shell*, or command interpreter, to start `awk` and use the *program* to process records in the input file(s). There are single quotes around *program* so that the shell doesn't interpret any `awk` characters as special shell characters. They also cause the shell to treat all of *program* as a single argument for `awk` and allow *program* to be more than one line long.

This format is also useful for running short or medium-sized `awk` programs from shell scripts, because it avoids the need for a separate file for the `awk` program. A self-contained shell script is more reliable since there are no other files to misplace.

Chapter 3 [Useful One Line Programs], page 21, presents several short, self-contained programs.

As an interesting side point, the command

```
awk '/foo/' files ...
```

is essentially the same as

```
egrep foo files ...
```

2.2.2 Running awk without Input Files

You can also run awk without any input files. If you type the command line:

```
awk 'program'
```

then awk applies the *program* to the *standard input*, which usually means whatever you type on the terminal. This continues until you indicate end-of-file by typing *Control-d*. (On other operating systems, the end-of-file character may be different. For example, on OS/2 and MS-DOS, it is *Control-z*.)

For example, the following program prints a friendly piece of advice (from Douglas Adams' *The Hitchhiker's Guide to the Galaxy*), to keep you from worrying about the complexities of computer programming ('BEGIN' is a feature we haven't discussed yet).

```
$ awk "BEGIN { print \"Don't Panic!\" }"
⊣ Don't Panic!
```

This program does not read any input. The '\' before each of the inner double quotes is necessary because of the shell's quoting rules, in particular because it mixes both single quotes and double quotes.

This next simple awk program emulates the cat utility; it copies whatever you type at the keyboard to its standard output. (Why this works is explained shortly.)

```
$ awk '{ print }'
Now is the time for all good men
⊣ Now is the time for all good men
to come to the aid of their country.
⊣ to come to the aid of their country.
Four score and seven years ago, ...
⊣ Four score and seven years ago, ...
What, me worry?
⊣ What, me worry?
Control-d
```

2.2.3 Running Long Programs

Sometimes your awk programs can be very long. In this case it is more convenient to put the program into a separate file. To tell awk to use that file for its program, you type:

```
awk -f source-file input-file1 input-file2 ...
```

The '-f' instructs the awk utility to get the awk program from the file *source-file*. Any file name can be used for *source-file*. For example, you could put the program:

```
BEGIN { print "Don't Panic!" }
```

into the file 'advice'. Then this command:

```
awk -f advice
```

does the same thing as this one:

```
awk "BEGIN { print \"Don't Panic!\" }"
```

which was explained earlier (see Section 2.2.2 [Running `awk` without Input Files], page 11). Note that you don't usually need single quotes around the file name that you specify with '`-f`', because most file names don't contain any of the shell's special characters. Notice that in '`advice`', the `awk` program did not have single quotes around it. The quotes are only needed for programs that are provided on the `awk` command line.

If you want to identify your `awk` program files clearly as such, you can add the extension '`.awk`' to the file name. This doesn't affect the execution of the `awk` program, but it does make "housekeeping" easier.

2.2.4 Executable `awk` Programs

Once you have learned `awk`, you may want to write self-contained `awk` scripts, using the '`#!`' script mechanism. You can do this on many Unix systems[2] (and someday on the GNU system).

For example, you could update the file '`advice`' to look like this:

```
#! /bin/awk -f

BEGIN    { print "Don't Panic!" }
```

After making this file executable (with the `chmod` utility), you can simply type '`advice`' at the shell, and the system will arrange to run `awk`[3] as if you had typed '`awk -f advice`'.

```
$ advice
⊣ Don't Panic!
```

Self-contained `awk` scripts are useful when you want to write a program which users can invoke without their having to know that the program is written in `awk`.

Caution: You should not put more than one argument on the '`#!`' line after the path to `awk`. This will not work. The operating system treats the rest of the line as a single agument, and passes it to `awk`. Doing this will lead to confusing behavior: most likely a usage diagnostic of some sort from `awk`.

---

[2] The '`#!`' mechanism works on Linux systems, Unix systems derived from Berkeley Unix, System V Release 4, and some System V Release 3 systems.

[3] The line beginning with '`#!`' lists the full file name of an interpreter to be run, and an optional initial command line argument to pass to that interpreter. The operating system then runs the interpreter with the given argument and the full argument list of the executed program. The first argument in the list is the full file name of the `awk` program. The rest of the argument list will either be options to `awk`, or data files, or both.

Some older systems do not support the '#!' mechanism. You can get a similar effect using a regular shell script. It would look something like this:

```
: The colon ensures execution by the standard shell.
awk 'program' "$@"
```

Using this technique, it is *vital* to enclose the *program* in single quotes to protect it from interpretation by the shell. If you omit the quotes, only a shell wizard can predict the results.

The "$@" causes the shell to forward all the command line arguments to the awk program, without interpretation. The first line, which starts with a colon, is used so that this shell script will work even if invoked by a user who uses the C shell. (Not all older systems obey this convention, but many do.)

2.2.5 Comments in awk Programs

A *comment* is some text that is included in a program for the sake of human readers; it is not really part of the program. Comments can explain what the program does, and how it works. Nearly all programming languages have provisions for comments, because programs are typically hard to understand without their extra help.

In the awk language, a comment starts with the sharp sign character, '#', and continues to the end of the line. The '#' does not have to be the first character on the line. The awk language ignores the rest of a line following a sharp sign. For example, we could have put the following into 'advice':

```
# This program prints a nice friendly message.  It helps
# keep novice users from being afraid of the computer.
BEGIN    { print "Don't Panic!" }
```

You can put comment lines into keyboard-composed throw-away awk programs also, but this usually isn't very useful; the purpose of a comment is to help you or another person understand the program at a later time.

Caution: As mentioned in Section 2.2.1 [One-shot Throw-away awk Programs], page 10, you can enclose small to medium programs in single quotes, in order to keep your shell scripts self-contained. When doing so, *don't* put an apostrophe (i.e., a single quote) into a comment (or anywhere else in your program). The shell will interpret the quote as the closing quote for the entire program. As a result, usually the shell will print a message about mismatched quotes, and if awk actually runs, it will probably print strange messages about syntax errors. For example:

```
awk 'BEGIN { print "hello" } # let's be cute'
```

2.3 A Very Simple Example

The following command runs a simple awk program that searches the input file 'BBS-list' for the string of characters: 'foo'. (A string of characters

is usually called a *string*. The term *string* is perhaps based on similar usage in English, such as "a string of pearls," or, "a string of cars in a train.")

```
awk '/foo/ { print $0 }' BBS-list
```

When lines containing '`foo`' are found, they are printed, because '`print $0`' means print the current line. (Just '`print`' by itself means the same thing, so we could have written that instead.)

You will notice that slashes, '`/`', surround the string '`foo`' in the `awk` program. The slashes indicate that '`foo`' is a pattern to search for. This type of pattern is called a *regular expression*, and is covered in more detail later (see Chapter 4 [Regular Expressions], page 23). The pattern is allowed to match parts of words. There are single-quotes around the `awk` program so that the shell won't interpret any of it as special shell characters.

Here is what this program prints:

```
$ awk '/foo/ { print $0 }' BBS-list
 ⊣ fooey        555-1234     2400/1200/300     B
 ⊣ foot         555-6699     1200/300          B
 ⊣ macfoo       555-6480     1200/300          A
 ⊣ sabafoo      555-2127     1200/300          C
```

In an `awk` rule, either the pattern or the action can be omitted, but not both. If the pattern is omitted, then the action is performed for *every* input line. If the action is omitted, the default action is to print all lines that match the pattern.

Thus, we could leave out the action (the `print` statement and the curly braces) in the above example, and the result would be the same: all lines matching the pattern '`foo`' would be printed. By comparison, omitting the `print` statement but retaining the curly braces makes an empty action that does nothing; then no lines would be printed.

2.4 An Example with Two Rules

The `awk` utility reads the input files one line at a time. For each line, `awk` tries the patterns of each of the rules. If several patterns match then several actions are run, in the order in which they appear in the `awk` program. If no patterns match, then no actions are run.

After processing all the rules (perhaps none) that match the line, `awk` reads the next line (however, see Section 9.7 [The `next` Statement], page 108, and also see Section 9.8 [The `nextfile` Statement], page 109). This continues until the end of the file is reached.

For example, the `awk` program:

```
/12/  { print $0 }
/21/  { print $0 }
```

contains two rules. The first rule has the string '`12`' as the pattern and '`print $0`' as the action. The second rule has the string '`21`' as the pattern

and also has '`print $0`' as the action. Each rule's action is enclosed in its own pair of braces.

This `awk` program prints every line that contains the string '`12`' *or* the string '`21`'. If a line contains both strings, it is printed twice, once by each rule.

This is what happens if we run this program on our two sample data files, '`BBS-list`' and '`inventory-shipped`', as shown here:

```
$ awk '/12/ { print $0 }
>      /21/ { print $0 }' BBS-list inventory-shipped
⊣ aardvark     555-5553     1200/300          B
⊣ alpo-net     555-3412     2400/1200/300     A
⊣ barfly       555-7685     1200/300          A
⊣ bites        555-1675     2400/1200/300     A
⊣ core         555-2912     1200/300          C
⊣ fooey        555-1234     2400/1200/300     B
⊣ foot         555-6699     1200/300          B
⊣ macfoo       555-6480     1200/300          A
⊣ sdace        555-3430     2400/1200/300     A
⊣ sabafoo      555-2127     1200/300          C
⊣ sabafoo      555-2127     1200/300          C
⊣ Jan  21  36  64 620
⊣ Apr  21  70  74 514
```

Note how the line in '`BBS-list`' beginning with '`sabafoo`' was printed twice, once for each rule.

2.5 A More Complex Example

Here is an example to give you an idea of what typical `awk` programs do. This example shows how `awk` can be used to summarize, select, and rearrange the output of another utility. It uses features that haven't been covered yet, so don't worry if you don't understand all the details.

```
ls -lg | awk '$6 == "Nov" { sum += $5 }
             END { print sum }'
```

This command prints the total number of bytes in all the files in the current directory that were last modified in November (of any year). (In the C shell you would need to type a semicolon and then a backslash at the end of the first line; in a POSIX-compliant shell, such as the Bourne shell or Bash, the GNU Bourne-Again shell, you can type the example as shown.)

The '`ls -lg`' part of this example is a system command that gives you a listing of the files in a directory, including file size and the date the file was last modified. Its output looks like this:

```
-rw-r--r--  1 arnold   user   1933 Nov  7 13:05 Makefile
-rw-r--r--  1 arnold   user  10809 Nov  7 13:03 gawk.h
```

```
-rw-r--r--  1 arnold   user    983 Apr 13 12:14 gawk.tab.h
-rw-r--r--  1 arnold   user  31869 Jun 15 12:20 gawk.y
-rw-r--r--  1 arnold   user  22414 Nov  7 13:03 gawk1.c
-rw-r--r--  1 arnold   user  37455 Nov  7 13:03 gawk2.c
-rw-r--r--  1 arnold   user  27511 Dec  9 13:07 gawk3.c
-rw-r--r--  1 arnold   user   7989 Nov  7 13:03 gawk4.c
```

The first field contains read-write permissions, the second field contains the number of links to the file, and the third field identifies the owner of the file. The fourth field identifies the group of the file. The fifth field contains the size of the file in bytes. The sixth, seventh and eighth fields contain the month, day, and time, respectively, that the file was last modified. Finally, the ninth field contains the name of the file.

The '`$6 == "Nov"`' in our `awk` program is an expression that tests whether the sixth field of the output from '`ls -lg`' matches the string '`Nov`'. Each time a line has the string '`Nov`' for its sixth field, the action '`sum += $5`' is performed. This adds the fifth field (the file size) to the variable `sum`. As a result, when `awk` has finished reading all the input lines, `sum` is the sum of the sizes of files whose lines matched the pattern. (This works because `awk` variables are automatically initialized to zero.)

After the last line of output from `ls` has been processed, the `END` rule is executed, and the value of `sum` is printed. In this example, the value of `sum` would be 80600.

These more advanced `awk` techniques are covered in later sections (see Section 8.2 [Overview of Actions], page 100). Before you can move on to more advanced `awk` programming, you have to know how `awk` interprets your input and displays your output. By manipulating fields and using `print` statements, you can produce some very useful and impressive looking reports.

2.6 `awk` Statements Versus Lines

Most often, each line in an `awk` program is a separate statement or separate rule, like this:

```
awk '/12/  { print $0 }
     /21/  { print $0 }' BBS-list inventory-shipped
```

However, `gawk` will ignore newlines after any of the following:

```
,    {    ?    :    ||    &&    do    else
```

A newline at any other point is considered the end of the statement. (Splitting lines after '`?`' and '`:`' is a minor `gawk` extension. The '`?`' and '`:`' referred to here is the three operand conditional expression described in Section 7.12 [Conditional Expressions], page 90.)

If you would like to split a single statement into two lines at a point where a newline would terminate it, you can *continue* it by ending the first line with a backslash character, '\'. The backslash must be the final character

on the line to be recognized as a continuation character. This is allowed absolutely anywhere in the statement, even in the middle of a string or regular expression. For example:

```
awk '/This regular expression is too long, so continue it\
 on the next line/ { print $1 }'
```

We have generally not used backslash continuation in the sample programs in this book. Since in gawk there is no limit on the length of a line, it is never strictly necessary; it just makes programs more readable. For this same reason, as well as for clarity, we have kept most statements short in the sample programs presented throughout the book. Backslash continuation is most useful when your awk program is in a separate source file, instead of typed in on the command line. You should also note that many awk implementations are more particular about where you may use backslash continuation. For example, they may not allow you to split a string constant using backslash continuation. Thus, for maximal portability of your awk programs, it is best not to split your lines in the middle of a regular expression or a string.

Caution: backslash continuation does not work as described above with the C shell. Continuation with backslash works for awk programs in files, and also for one-shot programs *provided* you are using a POSIX-compliant shell, such as the Bourne shell or Bash, the GNU Bourne-Again shell. But the C shell (csh) behaves differently! There, you must use two backslashes in a row, followed by a newline. Note also that when using the C shell, *every* newline in your awk program must be escaped with a backslash. To illustrate:

```
% awk 'BEGIN { \
?   print \\
?       "hello, world" \
? }'
⊣ hello, world
```

Here, the '%' and '?' are the C shell's primary and secondary prompts, analogous to the standard shell's '$' and '>'.

awk is a line-oriented language. Each rule's action has to begin on the same line as the pattern. To have the pattern and action on separate lines, you *must* use backslash continuation—there is no other way.

Note that backslash continuation and comments do not mix. As soon as awk sees the '#' that starts a comment, it ignores *everything* on the rest of the line. For example:

```
$ gawk 'BEGIN { print "dont panic" # a friendly \
>                                  BEGIN rule
> }'
error  gawk: cmd. line:2:              BEGIN rule
error  gawk: cmd. line:2:              ^ parse error
```

Here, it looks like the backslash would continue the comment onto the next line. However, the backslash-newline combination is never even noticed, since it is "hidden" inside the comment. Thus, the 'BEGIN' is noted as a syntax error.

When awk statements within one rule are short, you might want to put more than one of them on a line. You do this by separating the statements with a semicolon, ';'.

This also applies to the rules themselves. Thus, the previous program could have been written:

```
/12/ { print $0 } ; /21/ { print $0 }
```

Note: the requirement that rules on the same line must be separated with a semicolon was not in the original awk language; it was added for consistency with the treatment of statements within an action.

2.7 Other Features of awk

The awk language provides a number of predefined, or built-in variables, which your programs can use to get information from awk. There are other variables your program can set to control how awk processes your data.

In addition, awk provides a number of built-in functions for doing common computational and string related operations.

As we develop our presentation of the awk language, we introduce most of the variables and many of the functions. They are defined systematically in Chapter 10 [Built-in Variables], page 113, and Chapter 12 [Built-in Functions], page 133.

2.8 When to Use awk

You might wonder how awk might be useful for you. Using utility programs, advanced patterns, field separators, arithmetic statements, and other selection criteria, you can produce much more complex output. The awk language is very useful for producing reports from large amounts of raw data, such as summarizing information from the output of other utility programs like ls. (See Section 2.5 [A More Complex Example], page 15.)

Programs written with awk are usually much smaller than they would be in other languages. This makes awk programs easy to compose and use. Often, awk programs can be quickly composed at your terminal, used once, and thrown away. Since awk programs are interpreted, you can avoid the (usually lengthy) compilation part of the typical edit-compile-test-debug cycle of software development.

Complex programs have been written in awk, including a complete retargetable assembler for eight-bit microprocessors (see Appendix D [Glossary], page 295, for more information) and a microcode assembler for a special

purpose Prolog computer. However, `awk`'s capabilities are strained by tasks of such complexity.

If you find yourself writing `awk` scripts of more than, say, a few hundred lines, you might consider using a different programming language. Emacs Lisp is a good choice if you need sophisticated string or pattern matching capabilities. The shell is also good at string and pattern matching; in addition, it allows powerful use of the system utilities. More conventional languages, such as C, C++, and Lisp, offer better facilities for system programming and for managing the complexity of large programs. Programs in these languages may require more lines of source code than the equivalent `awk` programs, but they are easier to maintain and usually run more efficiently.

3 Useful One Line Programs

Many useful `awk` programs are short, just a line or two. Here is a collection of useful, short programs to get you started. Some of these programs contain constructs that haven't been covered yet. The description of the program will give you a good idea of what is going on, but please read the rest of the book to become an `awk` expert!

Most of the examples use a data file named '`data`'. This is just a placeholder; if you were to use these programs yourself, you would substitute your own file names for '`data`'.

```
awk '{ if (length($0) > max) max = length($0) }
     END { print max }' data
```
This program prints the length of the longest input line.

```
awk 'length($0) > 80' data
```
This program prints every line that is longer than 80 characters. The sole rule has a relational expression as its pattern, and has no action (so the default action, printing the record, is used).

```
expand data | awk '{ if (x < length()) x = length() }
                   END { print "maximum line length is " x }'
```
This program prints the length of the longest line in '`data`'. The input is processed by the `expand` program to change tabs into spaces, so the widths compared are actually the right-margin columns.

```
awk 'NF > 0' data
```
This program prints every line that has at least one field. This is an easy way to delete blank lines from a file (or rather, to create a new file similar to the old file but from which the blank lines have been deleted).

```
awk 'BEGIN { for (i = 1; i <= 7; i++)
                 print int(101 * rand()) }'
```
This program prints seven random numbers from zero to 100, inclusive.

```
ls -lg files | awk '{ x += $5 } ; END { print "total bytes: " x }'
```
This program prints the total number of bytes used by *files*.

```
ls -lg files | awk '{ x += $5 }
                   END { print "total K-bytes: " (x + 1023)/1024 }'
```
This program prints the total number of kilobytes used by *files*.

```
awk -F: '{ print $1 }' /etc/passwd | sort
```
This program prints a sorted list of the login names of all users.

```
awk 'END { print NR }' data
```
This program counts lines in a file.

`awk 'NR % 2 == 0' data`

This program prints the even numbered lines in the data file. If you were to use the expression '`NR % 2 == 1`' instead, it would print the odd numbered lines.

4 Regular Expressions

A *regular expression*, or *regexp*, is a way of describing a set of strings. Because regular expressions are such a fundamental part of `awk` programming, their format and use deserve a separate chapter.

A regular expression enclosed in slashes (‘`/`’) is an `awk` pattern that matches every input record whose text belongs to that set.

The simplest regular expression is a sequence of letters, numbers, or both. Such a regexp matches any string that contains that sequence. Thus, the regexp ‘`foo`’ matches any string containing ‘`foo`’. Therefore, the pattern `/foo/` matches any input record containing the three characters ‘`foo`’, *anywhere* in the record. Other kinds of regexps let you specify more complicated classes of strings.

Initially, the examples will be simple. As we explain more about how regular expressions work, we will present more complicated examples.

4.1 How to Use Regular Expressions

A regular expression can be used as a pattern by enclosing it in slashes. Then the regular expression is tested against the entire text of each record. (Normally, it only needs to match some part of the text in order to succeed.) For example, this prints the second field of each record that contains the three characters ‘`foo`’ anywhere in it:

```
$ awk '/foo/ { print $2 }' BBS-list
⊣ 555-1234
⊣ 555-6699
⊣ 555-6480
⊣ 555-2127
```

Regular expressions can also be used in matching expressions. These expressions allow you to specify the string to match against; it need not be the entire current input record. The two operators, ‘`~`’ and ‘`!~`’, perform regular expression comparisons. Expressions using these operators can be used as patterns or in `if`, `while`, `for`, and `do` statements.

exp `~` */regexp/*

This is true if the expression *exp* (taken as a string) is matched by *regexp*. The following example matches, or selects, all input records with the upper-case letter ‘`J`’ somewhere in the first field:

```
$ awk '$1 ~ /J/' inventory-shipped
⊣ Jan  13  25  15 115
⊣ Jun  31  42  75 492
⊣ Jul  24  34  67 436
⊣ Jan  21  36  64 620
```

So does this:

```
awk '{ if ($1 ~ /J/) print }' inventory-shipped
```

exp `!~ /`*regexp*`/`
: This is true if the expression *exp* (taken as a character string) is *not* matched by *regexp*. The following example matches, or selects, all input records whose first field *does not* contain the upper-case letter 'J':

```
$ awk '$1 !~ /J/' inventory-shipped
⊣ Feb  15  32  24 226
⊣ Mar  15  24  34 228
⊣ Apr  31  52  63 420
⊣ May  16  34  29 208
...
```

When a regexp is written enclosed in slashes, like `/foo/`, we call it a *regexp constant*, much like `5.27` is a numeric constant, and `"foo"` is a string constant.

4.2 Escape Sequences

Some characters cannot be included literally in string constants (`"foo"`) or regexp constants (`/foo/`). You represent them instead with *escape sequences*, which are character sequences beginning with a backslash ('\').

One use of an escape sequence is to include a double-quote character in a string constant. Since a plain double-quote would end the string, you must use '`\"`' to represent an actual double-quote character as a part of the string. For example:

```
$ awk 'BEGIN { print "He said \"hi!\" to her." }'
⊣ He said "hi!" to her.
```

The backslash character itself is another character that cannot be included normally; you write '`\\`' to put one backslash in the string or regexp. Thus, the string whose contents are the two characters '"' and '\' must be written `"\"\\"`.

Another use of backslash is to represent unprintable characters such as tab or newline. While there is nothing to stop you from entering most unprintable characters directly in a string constant or regexp constant, they may look ugly.

Here is a table of all the escape sequences used in `awk`, and what they represent. Unless noted otherwise, all of these escape sequences apply to both string constants and regexp constants.

`\\`
: A literal backslash, '\'.

`\a`
: The "alert" character, *`Control-g`*, ASCII code 7 (BEL).

`\b`
: Backspace, *`Control-h`*, ASCII code 8 (BS).

`\f`	Formfeed, *`Control-l`*, ASCII code 12 (FF).
`\n`	Newline, *`Control-j`*, ASCII code 10 (LF).
`\r`	Carriage return, *`Control-m`*, ASCII code 13 (CR).
`\t`	Horizontal tab, *`Control-i`*, ASCII code 9 (HT).
`\v`	Vertical tab, *`Control-k`*, ASCII code 11 (VT).
`\`*nnn*	The octal value *nnn*, where *nnn* are one to three digits between '`0`' and '`7`'. For example, the code for the ASCII ESC (escape) character is '`\033`'.
`\x`*hh*...	The hexadecimal value *hh*, where *hh* are hexadecimal digits ('`0`' through '`9`' and either '`A`' through '`F`' or '`a`' through '`f`'). Like the same construct in ANSI C, the escape sequence continues until the first non-hexadecimal digit is seen. However, using more than two hexadecimal digits produces undefined results. (The '`\x`' escape sequence is not allowed in POSIX `awk`.)
`\/`	A literal slash (necessary for regexp constants only). You use this when you wish to write a regexp constant that contains a slash. Since the regexp is delimited by slashes, you need to escape the slash that is part of the pattern, in order to tell `awk` to keep processing the rest of the regexp.
`\"`	A literal double-quote (necessary for string constants only). You use this when you wish to write a string constant that contains a double-quote. Since the string is delimited by double-quotes, you need to escape the quote that is part of the string, in order to tell `awk` to keep processing the rest of the string.

In `gawk`, there are additional two character sequences that begin with backslash that have special meaning in regexps. See Section 4.4 [Additional Regexp Operators Only in `gawk`], page 31.

In a string constant, what happens if you place a backslash before something that is not one of the characters listed above? POSIX `awk` purposely leaves this case undefined. There are two choices.

- Strip the backslash out. This is what Unix `awk` and `gawk` both do. For example, `"a\qc"` is the same as `"aqc"`.
- Leave the backslash alone. Some other `awk` implementations do this. In such implementations, `"a\qc"` is the same as if you had typed `"a\\qc"`.

In a regexp, a backslash before any character that is not in the above table, and not listed in Section 4.4 [Additional Regexp Operators Only in `gawk`], page 31, means that the next character should be taken literally, even if it would normally be a regexp operator. E.g., `/a\+b/` matches the three characters '`a+b`'.

For complete portability, do not use a backslash before any character not listed in the table above.

Another interesting question arises. Suppose you use an octal or hexadecimal escape to represent a regexp metacharacter (see Section 4.3 [Regular Expression Operators], page 26). Does `awk` treat the character as a literal character, or as a regexp operator?

It turns out that historically, such characters were taken literally (d.c.). However, the POSIX standard indicates that they should be treated as real metacharacters, and this is what `gawk` does. However, in compatibility mode (see Section 14.1 [Command Line Options], page 159), `gawk` treats the characters represented by octal and hexadecimal escape sequences literally when used in regexp constants. Thus, `/a\52b/` is equivalent to `/a\*b/`.

To summarize:

1. The escape sequences in the table above are always processed first, for both string constants and regexp constants. This happens very early, as soon as `awk` reads your program.
2. `gawk` processes both regexp constants and dynamic regexps (see Section 4.7 [Using Dynamic Regexps], page 35), for the special operators listed in Section 4.4 [Additional Regexp Operators Only in `gawk`], page 31.
3. A backslash before any other character means to treat that character literally.

4.3 Regular Expression Operators

You can combine regular expressions with the following characters, called *regular expression operators*, or *metacharacters*, to increase the power and versatility of regular expressions.

The escape sequences described above in Section 4.2 [Escape Sequences], page 24, are valid inside a regexp. They are introduced by a '\'. They are recognized and converted into the corresponding real characters as the very first step in processing regexps.

Here is a table of metacharacters. All characters that are not escape sequences and that are not listed in the table stand for themselves.

`\` This is used to suppress the special meaning of a character when matching. For example:

```
\$
```

matches the character '`$`'.

`^` This matches the beginning of a string. For example:

```
^@chapter
```

matches the '`@chapter`' at the beginning of a string, and can be used to identify chapter beginnings in Texinfo source files. The '`^`' is known as an *anchor*, since it anchors the pattern to matching only at the beginning of the string.

It is important to realize that '`^`' does not match the beginning of a line embedded in a string. In this example the condition is not true:

```
if ("line1\nLINE 2" ~ /^L/) ...
```

`$` This is similar to '`^`', but it matches only at the end of a string. For example:

```
p$
```

matches a record that ends with a '`p`'. The '`$`' is also an anchor, and also does not match the end of a line embedded in a string. In this example the condition is not true:

```
if ("line1\nLINE 2" ~ /1$/) ...
```

`.` The period, or dot, matches any single character, *including* the newline character. For example:

```
.P
```

matches any single character followed by a '`P`' in a string. Using concatenation we can make a regular expression like '`U.A`', which matches any three-character sequence that begins with '`U`' and ends with '`A`'.

In strict POSIX mode (see Section 14.1 [Command Line Options], page 159), '`.`' does not match the NUL character, which is a character with all bits equal to zero. Otherwise, NUL is just another character. Other versions of `awk` may not be able to match the NUL character.

`[...]` This is called a *character list*. It matches any *one* of the characters that are enclosed in the square brackets. For example:

```
[MVX]
```

matches any one of the characters '`M`', '`V`', or '`X`' in a string.

Ranges of characters are indicated by using a hyphen between the beginning and ending characters, and enclosing the whole thing in brackets. For example:

```
[0-9]
```

matches any digit. Multiple ranges are allowed. E.g., the list `[A-Za-z0-9]` is a common way to express the idea of "all alphanumeric characters."

To include one of the characters '\', ']', '-' or '^' in a character list, put a '\' in front of it. For example:

```
[d\]]
```

matches either 'd', or ']'.

This treatment of '\' in character lists is compatible with other `awk` implementations, and is also mandated by POSIX. The regular expressions in `awk` are a superset of the POSIX specification for Extended Regular Expressions (EREs). POSIX EREs are based on the regular expressions accepted by the traditional `egrep` utility.

Character classes are a new feature introduced in the POSIX standard. A character class is a special notation for describing lists of characters that have a specific attribute, but where the actual characters themselves can vary from country to country and/or from character set to character set. For example, the notion of what is an alphabetic character differs in the USA and in France.

A character class is only valid in a regexp *inside* the brackets of a character list. Character classes consist of '`[:`', a keyword denoting the class, and '`:]`'. Here are the character classes defined by the POSIX standard.

`[:alnum:]`
: Alphanumeric characters.

`[:alpha:]`
: Alphabetic characters.

`[:blank:]`
: Space and tab characters.

`[:cntrl:]`
: Control characters.

`[:digit:]`
: Numeric characters.

`[:graph:]`
: Characters that are printable and are also visible. (A space is printable, but not visible, while an 'a' is both.)

`[:lower:]`
: Lower-case alphabetic characters.

`[:print:]`
: Printable characters (characters that are not control characters.)

`[:punct:]`
: Punctuation characters (characters that are not letter, digits, control characters, or space characters).

`[:space:]`
: Space characters (such as space, tab, and formfeed, to name a few).

`[:upper:]`
: Upper-case alphabetic characters.

`[:xdigit:]`
: Characters that are hexadecimal digits.

For example, before the POSIX standard, to match alphanumeric characters, you had to write `/[A-Za-z0-9]/`. If your character set had other alphabetic characters in it, this would not match them. With the POSIX character classes, you can write `/[[:alnum:]]/`, and this will match *all* the alphabetic and numeric characters in your character set.

Two additional special sequences can appear in character lists. These apply to non-ASCII character sets, which can have single symbols (called *collating elements*) that are represented with more than one character, as well as several characters that are equivalent for *collating*, or sorting, purposes. (E.g., in French, a plain "e" and a grave-accented "è" are equivalent.)

Collating Symbols
: A *collating symbol* is a multi-character collating element enclosed in '`[.`' and '`.]`'. For example, if '`ch`' is a collating element, then `[[.ch.]]` is a regexp that matches this collating element, while `[ch]` is a regexp that matches either '`c`' or '`h`'.

Equivalence Classes
: An *equivalence class* is a locale-specific name for a list of characters that are equivalent. The name is enclosed in '`[=`' and '`=]`'. For example, the name '`e`' might be used to represent all of "e," "è," and "é." In this case, `[[=e]]` is a regexp that matches any of '`e`', '`é`', or '`è`'.

These features are very valuable in non-English speaking locales.

Caution: The library functions that `gawk` uses for regular expression matching currently only recognize POSIX character classes; they do not recognize collating symbols or equivalence classes.

`[^ ...]`
: This is a *complemented character list*. The first character after the '`[`' *must* be a '`^`'. It matches any characters *except* those in the square brackets. For example:

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

matches any character that is not a digit.

`|` This is the *alternation operator*, and it is used to specify alternatives. For example:

```
^P|[0-9]
```

matches any string that matches either '`^P`' or '`[0-9]`'. This means it matches any string that starts with '`P`' or contains a digit.

The alternation applies to the largest possible regexps on either side. In other words, '`|`' has the lowest precedence of all the regular expression operators.

`(...)` Parentheses are used for grouping in regular expressions as in arithmetic. They can be used to concatenate regular expressions containing the alternation operator, '`|`'. For example, '`@(samp|code)\{[^}]+\}`' matches both '`@code{foo}`' and '`@samp{bar}`'. (These are Texinfo formatting control sequences.)

`*` This symbol means that the preceding regular expression is to be repeated as many times as necessary to find a match. For example:

```
ph*
```

applies the '`*`' symbol to the preceding '`h`' and looks for matches of one '`p`' followed by any number of '`h`'s. This will also match just '`p`' if no '`h`'s are present.

The '`*`' repeats the *smallest* possible preceding expression. (Use parentheses if you wish to repeat a larger expression.) It finds as many repetitions as possible. For example:

```
awk '/\(c[ad][ad]*r x\)/ { print }' sample
```

prints every record in '`sample`' containing a string of the form '`(car x)`', '`(cdr x)`', '`(cadr x)`', and so on. Notice the escaping of the parentheses by preceding them with backslashes.

`+` This symbol is similar to '`*`', but the preceding expression must be matched at least once. This means that:

```
wh+y
```

would match '`why`' and '`whhy`' but not '`wy`', whereas '`wh*y`' would match all three of these strings. This is a simpler way of writing the last '`*`' example:

```
awk '/\(c[ad]+r x\)/ { print }' sample
```

`?` This symbol is similar to '`*`', but the preceding expression can be matched either once or not at all. For example:

```
fe?d
```

will match '`fed`' and '`fd`', but nothing else.

`{`*n*`}`
`{`*n*`,}`
`{`*n*`,`*m*`}` One or two numbers inside braces denote an *interval expression*. If there is one number in the braces, the preceding regexp is repeated *n* times. If there are two numbers separated by a comma, the preceding regexp is repeated *n* to *m* times. If there is one number followed by a comma, then the preceding regexp is repeated at least *n* times.

`wh{3}y` matches '`whhhy`' but not '`why`' or '`whhhhy`'.

`wh{3,5}y` matches '`whhhy`' or '`whhhhy`' or '`whhhhhy`', only.

`wh{2,}y` matches '`whhy`' or '`whhhy`', and so on.

Interval expressions were not traditionally available in `awk`. As part of the POSIX standard they were added, to make `awk` and `egrep` consistent with each other.

However, since old programs may use '`{`' and '`}`' in regexp constants, by default `gawk` does *not* match interval expressions in regexps. If either '`--posix`' or '`--re-interval`' are specified (see Section 14.1 [Command Line Options], page 159), then interval expressions are allowed in regexps.

In regular expressions, the '`*`', '`+`', and '`?`' operators, as well as the braces '`{`' and '`}`', have the highest precedence, followed by concatenation, and finally by '`|`'. As in arithmetic, parentheses can change how operators are grouped.

If `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159), character classes and interval expressions are not available in regular expressions.

The next section discusses the GNU-specific regexp operators, and provides more detail concerning how command line options affect the way `gawk` interprets the characters in regular expressions.

4.4 Additional Regexp Operators Only in `gawk`

GNU software that deals with regular expressions provides a number of additional regexp operators. These operators are described in this section, and are specific to `gawk`; they are not available in other `awk` implementations.

Most of the additional operators are for dealing with word matching. For our purposes, a *word* is a sequence of one or more letters, digits, or underscores ('`_`').

`\w` This operator matches any word-constituent character, i.e. any letter, digit, or underscore. Think of it as a short-hand for `[[:alnum:]_]`.

`\W` This operator matches any character that is not word-constituent. Think of it as a short-hand for `[^[:alnum:]_]`.

`\<` This operator matches the empty string at the beginning of a word. For example, `/\<away/` matches '`away`', but not '`stowaway`'.

`\>` This operator matches the empty string at the end of a word. For example, `/stow\>/` matches '`stow`', but not '`stowaway`'.

`\y` This operator matches the empty string at either the beginning or the end of a word (the word boundary). For example, '`\yballs?\y`' matches either '`ball`' or '`balls`' as a separate word.

`\B` This operator matches the empty string within a word. In other words, '`\B`' matches the empty string that occurs between two word-constituent characters. For example, `/\Brat\B/` matches '`crate`', but it does not match '`dirty rat`'. '`\B`' is essentially the opposite of '`\y`'.

There are two other operators that work on buffers. In Emacs, a *buffer* is, naturally, an Emacs buffer. For other programs, the regexp library routines that `gawk` uses consider the entire string to be matched as the buffer.

For `awk`, since '`^`' and '`$`' always work in terms of the beginning and end of strings, these operators don't add any new capabilities. They are provided for compatibility with other GNU software.

`\‘` This operator matches the empty string at the beginning of the buffer.

`\’` This operator matches the empty string at the end of the buffer.

In other GNU software, the word boundary operator is '`\b`'. However, that conflicts with the `awk` language's definition of '`\b`' as backspace, so `gawk` uses a different letter.

An alternative method would have been to require two backslashes in the GNU operators, but this was deemed to be too confusing, and the current method of using '`\y`' for the GNU '`\b`' appears to be the lesser of two evils.

The various command line options (see Section 14.1 [Command Line Options], page 159) control how `gawk` interprets characters in regexps.

No options
: In the default case, `gawk` provides all the facilities of POSIX regexps and the GNU regexp operators described above. However, interval expressions are not supported.

`--posix` Only POSIX regexps are supported, the GNU operators are not special (e.g., '`\w`' matches a literal '`w`'). Interval expressions are allowed.

`--traditional`
: Traditional Unix `awk` regexps are matched. The GNU operators are not special, interval expressions are not available, and neither are the POSIX character classes (`[[:alnum:]]` and so on). Characters described by octal and hexadecimal escape sequences are treated literally, even if they represent regexp metacharacters.

`--re-interval`
: Allow interval expressions in regexps, even if '`--traditional`' has been provided.

4.5 Case-sensitivity in Matching

Case is normally significant in regular expressions, both when matching ordinary characters (i.e. not metacharacters), and inside character sets. Thus a '`w`' in a regular expression matches only a lower-case '`w`' and not an upper-case '`W`'.

The simplest way to do a case-independent match is to use a character list: '`[Ww]`'. However, this can be cumbersome if you need to use it often; and it can make the regular expressions harder to read. There are two alternatives that you might prefer.

One way to do a case-insensitive match at a particular point in the program is to convert the data to a single case, using the `tolower` or `toupper` built-in string functions (which we haven't discussed yet; see Section 12.3 [Built-in Functions for String Manipulation], page 135). For example:

```
tolower($1) ~ /foo/  { ... }
```

converts the first field to lower-case before matching against it. This will work in any POSIX-compliant implementation of `awk`.

Another method, specific to `gawk`, is to set the variable `IGNORECASE` to a non-zero value (see Chapter 10 [Built-in Variables], page 113). When `IGNORECASE` is not zero, *all* regexp and string operations ignore case. Changing the value of `IGNORECASE` dynamically controls the case sensitivity of your program as it runs. Case is significant by default because `IGNORECASE` (like most variables) is initialized to zero.

```
x = "aB"
if (x ~ /ab/) ...   # this test will fail

IGNORECASE = 1
if (x ~ /ab/) ...   # now it will succeed
```

In general, you cannot use `IGNORECASE` to make certain rules case-insensitive and other rules case-sensitive, because there is no way to set `IGNORECASE` just for the pattern of a particular rule. To do this, you must use character lists or `tolower`. However, one thing you can do only with

`IGNORECASE` is turn case-sensitivity on or off dynamically for all the rules at once.

`IGNORECASE` can be set on the command line, or in a `BEGIN` rule (see Section 14.2 [Other Command Line Arguments], page 163; also see Section 8.1.5.1 [Startup and Cleanup Actions], page 98). Setting `IGNORECASE` from the command line is a way to make a program case-insensitive without having to edit it.

Prior to version 3.0 of `gawk`, the value of `IGNORECASE` only affected regexp operations. It did not affect string comparison with '`==`', '`!=`', and so on. Beginning with version 3.0, both regexp and string comparison operations are affected by `IGNORECASE`.

Beginning with version 3.0 of `gawk`, the equivalences between upper-case and lower-case characters are based on the ISO-8859-1 (ISO Latin-1) character set. This character set is a superset of the traditional 128 ASCII characters, that also provides a number of characters suitable for use with European languages.

The value of `IGNORECASE` has no effect if `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159). Case is always significant in compatibility mode.

4.6 How Much Text Matches?

Consider the following example:

```
echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
```

This example uses the `sub` function (which we haven't discussed yet, see Section 12.3 [Built-in Functions for String Manipulation], page 135) to make a change to the input record. Here, the regexp `/a+/` indicates "one or more '`a`' characters," and the replacement text is '`<A>`'.

The input contains four '`a`' characters. What will the output be? In other words, how many is "one or more"—will `awk` match two, three, or all four '`a`' characters?

The answer is, `awk` (and POSIX) regular expressions always match the leftmost, *longest* sequence of input characters that can match. Thus, in this example, all four '`a`' characters are replaced with '`<A>`'.

```
$ echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
⊣ <A>bcd
```

For simple match/no-match tests, this is not so important. But when doing text matching and substitutions with the `match`, `sub`, `gsub`, and `gensub` functions, it is very important. Understanding this principle is also important for regexp-based record and field splitting (see Section 5.1 [How Input is Split into Records], page 37, and also see Section 5.5 [Specifying How Fields are Separated], page 44).

4.7 Using Dynamic Regexps

The right hand side of a '~' or '!~' operator need not be a regexp constant (i.e. a string of characters between slashes). It may be any expression. The expression is evaluated, and converted if necessary to a string; the contents of the string are used as the regexp. A regexp that is computed in this way is called a *dynamic regexp*. For example:

```
BEGIN { identifier_regexp = "[A-Za-z_][A-Za-z_0-9]*" }
$0 ~ identifier_regexp    { print }
```

sets `identifier_regexp` to a regexp that describes `awk` variable names, and tests if the input record matches this regexp.

Caution: When using the '~' and '!~' operators, there is a difference between a regexp constant enclosed in slashes, and a string constant enclosed in double quotes. If you are going to use a string constant, you have to understand that the string is in essence scanned *twice*; the first time when `awk` reads your program, and the second time when it goes to match the string on the left-hand side of the operator with the pattern on the right. This is true of any string valued expression (such as `identifier_regexp` above), not just string constants.

What difference does it make if the string is scanned twice? The answer has to do with escape sequences, and particularly with backslashes. To get a backslash into a regular expression inside a string, you have to type two backslashes.

For example, `/\*/` is a regexp constant for a literal '*'. Only one backslash is needed. To do the same thing with a string, you would have to type `"\\*"`. The first backslash escapes the second one, so that the string actually contains the two characters '\' and '*'.

Given that you can use both regexp and string constants to describe regular expressions, which should you use? The answer is "regexp constants," for several reasons.

1. String constants are more complicated to write, and more difficult to read. Using regexp constants makes your programs less error-prone. Not understanding the difference between the two kinds of constants is a common source of errors.
2. It is also more efficient to use regexp constants: `awk` can note that you have supplied a regexp and store it internally in a form that makes pattern matching more efficient. When using a string constant, `awk` must first convert the string into this internal form, and then perform the pattern matching.
3. Using regexp constants is better style; it shows clearly that you intend a regexp match.

5 Reading Input Files

In the typical `awk` program, all input is read either from the standard input (by default the keyboard, but often a pipe from another command) or from files whose names you specify on the `awk` command line. If you specify input files, `awk` reads them in order, reading all the data from one before going on to the next. The name of the current input file can be found in the built-in variable `FILENAME` (see Chapter 10 [Built-in Variables], page 113).

The input is read in units called *records*, and processed by the rules of your program one record at a time. By default, each record is one line. Each record is automatically split into chunks called *fields*. This makes it more convenient for programs to work on the parts of a record.

On rare occasions you will need to use the `getline` command. The `getline` command is valuable, both because it can do explicit input from any number of files, and because the files used with it do not have to be named on the `awk` command line (see Section 5.8 [Explicit Input with `getline`], page 53).

5.1 How Input is Split into Records

The `awk` utility divides the input for your `awk` program into records and fields. Records are separated by a character called the *record separator*. By default, the record separator is the newline character. This is why records are, by default, single lines. You can use a different character for the record separator by assigning the character to the built-in variable `RS`.

You can change the value of `RS` in the `awk` program, like any other variable, with the assignment operator, '`=`' (see Section 7.7 [Assignment Expressions], page 82). The new record-separator character should be enclosed in quotation marks, which indicate a string constant. Often the right time to do this is at the beginning of execution, before any input has been processed, so that the very first record will be read with the proper separator. To do this, use the special `BEGIN` pattern (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98). For example:

```
awk 'BEGIN { RS = "/" } ; { print $0 }' BBS-list
```

changes the value of `RS` to `"/"`, before reading any input. This is a string whose first character is a slash; as a result, records are separated by slashes. Then the input file is read, and the second rule in the `awk` program (the action with no pattern) prints each record. Since each `print` statement adds a newline at the end of its output, the effect of this `awk` program is to copy the input with each slash changed to a newline. Here are the results of running the program on '`BBS-list`':

```
$ awk 'BEGIN { RS = "/" } ; { print $0 }' BBS-list
⊣ aardvark     555-5553     1200
⊣ 300          B
⊣ alpo-net     555-3412     2400
⊣ 1200
⊣ 300     A
⊣ barfly       555-7685     1200
⊣ 300          A
⊣ bites        555-1675     2400
⊣ 1200
⊣ 300     A
⊣ camelot      555-0542     300               C
⊣ core         555-2912     1200
⊣ 300          C
⊣ fooey        555-1234     2400
⊣ 1200
⊣ 300     B
⊣ foot         555-6699     1200
⊣ 300          B
⊣ macfoo       555-6480     1200
⊣ 300          A
⊣ sdace        555-3430     2400
⊣ 1200
⊣ 300     A
⊣ sabafoo      555-2127     1200
⊣ 300          C
⊣
```

Note that the entry for the '`camelot`' BBS is not split. In the original data file (see Section 1.3 [Data Files for the Examples], page 7), the line looks like this:

```
camelot      555-0542     300               C
```

It only has one baud rate; there are no slashes in the record.

Another way to change the record separator is on the command line, using the variable-assignment feature (see Section 14.2 [Other Command Line Arguments], page 163).

```
awk '{ print $0 }' RS="/" BBS-list
```

This sets `RS` to '`/`' before processing '`BBS-list`'.

Using an unusual character such as '`/`' for the record separator produces correct behavior in the vast majority of cases. However, the following (extreme) pipeline prints a surprising '`1`'. There is one field, consisting of a newline. The value of the built-in variable `NF` is the number of fields in the current record.

```
$ echo | awk 'BEGIN { RS = "a" } ; { print NF }'
⊣ 1
```

Reaching the end of an input file terminates the current input record, even if the last character in the file is not the character in `RS` (d.c.).

The empty string, `""` (a string of no characters), has a special meaning as the value of `RS`: it means that records are separated by one or more blank lines, and nothing else. See Section 5.7 [Multiple-Line Records], page 50, for more details.

If you change the value of `RS` in the middle of an `awk` run, the new value is used to delimit subsequent records, but the record currently being processed (and records already processed) are not affected.

After the end of the record has been determined, `gawk` sets the variable `RT` to the text in the input that matched `RS`.

The value of `RS` is in fact not limited to a one-character string. It can be any regular expression (see Chapter 4 [Regular Expressions], page 23). In general, each record ends at the next string that matches the regular expression; the next record starts at the end of the matching string. This general rule is actually at work in the usual case, where `RS` contains just a newline: a record ends at the beginning of the next matching string (the next newline in the input) and the following record starts just after the end of this string (at the first character of the following line). The newline, since it matches `RS`, is not part of either record.

When `RS` is a single character, `RT` will contain the same single character. However, when `RS` is a regular expression, then `RT` becomes more useful; it contains the actual input text that matched the regular expression.

The following example illustrates both of these features. It sets `RS` equal to a regular expression that matches either a newline, or a series of one or more upper-case letters with optional leading and/or trailing white space (see Chapter 4 [Regular Expressions], page 23).

```
$ echo record 1 AAAA record 2 BBBB record 3 |
> gawk 'BEGIN { RS = "\n|( *[[:upper:]]+ *)" }
>             { print "Record =", $0, "and RT =", RT }'
⊣ Record = record 1 and RT =  AAAA
⊣ Record = record 2 and RT =  BBBB
⊣ Record = record 3 and RT =
⊣
```

The final line of output has an extra blank line. This is because the value of `RT` is a newline, and then the `print` statement supplies its own terminating newline.

See Section 16.2.8 [A Simple Stream Editor], page 237, for a more useful example of `RS` as a regexp and `RT`.

The use of `RS` as a regular expression and the `RT` variable are `gawk` extensions; they are not available in compatibility mode (see Section 14.1 [Com-

mand Line Options], page 159). In compatibility mode, only the first character of the value of `RS` is used to determine the end of the record.

The `awk` utility keeps track of the number of records that have been read so far from the current input file. This value is stored in a built-in variable called `FNR`. It is reset to zero when a new file is started. Another built-in variable, `NR`, is the total number of input records read so far from all data files. It starts at zero but is never automatically reset to zero.

5.2 Examining Fields

When `awk` reads an input record, the record is automatically separated or *parsed* by the interpreter into chunks called *fields*. By default, fields are separated by whitespace, like words in a line. Whitespace in `awk` means any string of one or more spaces, tabs or newlines;[1] other characters such as formfeed, and so on, that are considered whitespace by other languages are *not* considered whitespace by `awk`.

The purpose of fields is to make it more convenient for you to refer to these pieces of the record. You don't have to use them—you can operate on the whole record if you wish—but fields are what make simple `awk` programs so powerful.

To refer to a field in an `awk` program, you use a dollar-sign, '`$`', followed by the number of the field you want. Thus, `$1` refers to the first field, `$2` to the second, and so on. For example, suppose the following is a line of input:

```
This seems like a pretty nice example.
```

Here the first field, or `$1`, is '`This`'; the second field, or `$2`, is '`seems`'; and so on. Note that the last field, `$7`, is '`example.`'. Because there is no space between the '`e`' and the '`.`', the period is considered part of the seventh field.

`NF` is a built-in variable whose value is the number of fields in the current record. `awk` updates the value of `NF` automatically, each time a record is read.

No matter how many fields there are, the last field in a record can be represented by `$NF`. So, in the example above, `$NF` would be the same as `$7`, which is '`example.`'. Why this works is explained below (see Section 5.3 [Non-constant Field Numbers], page 41). If you try to reference a field beyond the last one, such as `$8` when the record has only seven fields, you get the empty string.

`$0`, which looks like a reference to the "zeroth" field, is a special case: it represents the whole input record. `$0` is used when you are not interested in fields.

---

[1] In POSIX `awk`, newlines are not considered whitespace for separating fields.

Here are some more examples:

```
$ awk '$1 ~ /foo/ { print $0 }' BBS-list
 ⊣ fooey        555-1234     2400/1200/300     B
 ⊣ foot         555-6699     1200/300          B
 ⊣ macfoo       555-6480     1200/300          A
 ⊣ sabafoo      555-2127     1200/300          C
```

This example prints each record in the file '`BBS-list`' whose first field contains the string '`foo`'. The operator '`~`' is called a *matching operator* (see Section 4.1 [How to Use Regular Expressions], page 23); it tests whether a string (here, the field `$1`) matches a given regular expression.

By contrast, the following example looks for '`foo`' in *the entire record* and prints the first field and the last field for each input record containing a match.

```
$ awk '/foo/ { print $1, $NF }' BBS-list
 ⊣ fooey B
 ⊣ foot B
 ⊣ macfoo A
 ⊣ sabafoo C
```

5.3 Non-constant Field Numbers

The number of a field does not need to be a constant. Any expression in the `awk` language can be used after a '`$`' to refer to a field. The value of the expression specifies the field number. If the value is a string, rather than a number, it is converted to a number. Consider this example:

```
awk '{ print $NR }'
```

Recall that `NR` is the number of records read so far: one in the first record, two in the second, etc. So this example prints the first field of the first record, the second field of the second record, and so on. For the twentieth record, field number 20 is printed; most likely, the record has fewer than 20 fields, so this prints a blank line.

Here is another example of using expressions as field numbers:

```
awk '{ print $(2*2) }' BBS-list
```

`awk` must evaluate the expression '`(2*2)`' and use its value as the number of the field to print. The '`*`' sign represents multiplication, so the expression '`2*2`' evaluates to four. The parentheses are used so that the multiplication is done before the '`$`' operation; they are necessary whenever there is a binary operator in the field-number expression. This example, then, prints the hours of operation (the fourth field) for every line of the file '`BBS-list`'. (All of the `awk` operators are listed, in order of decreasing precedence, in Section 7.14 [Operator Precedence (How Operators Nest)], page 92.)

If the field number you compute is zero, you get the entire record. Thus, `$(2-2)` has the same value as `$0`. Negative field numbers are not allowed;

trying to reference one will usually terminate your running `awk` program. (The POSIX standard does not define what happens when you reference a negative field number. `gawk` will notice this and terminate your program. Other `awk` implementations may behave differently.)

As mentioned in Section 5.2 [Examining Fields], page 40, the number of fields in the current record is stored in the built-in variable `NF` (also see Chapter 10 [Built-in Variables], page 113). The expression `$NF` is not a special feature: it is the direct consequence of evaluating `NF` and using its value as a field number.

5.4 Changing the Contents of a Field

You can change the contents of a field as seen by `awk` within an `awk` program; this changes what `awk` perceives as the current input record. (The actual input is untouched; `awk` *never* modifies the input file.)

Consider this example and its output:

```
$ awk '{ $3 = $2 - 10; print $2, $3 }' inventory-shipped
⊣ 13 3
⊣ 15 5
⊣ 15 5
...
```

The '`-`' sign represents subtraction, so this program reassigns field three, `$3`, to be the value of field two minus ten, '`$2 - 10`'. (See Section 7.5 [Arithmetic Operators], page 80.) Then field two, and the new value for field three, are printed.

In order for this to work, the text in field `$2` must make sense as a number; the string of characters must be converted to a number in order for the computer to do arithmetic on it. The number resulting from the subtraction is converted back to a string of characters which then becomes field three. See Section 7.4 [Conversion of Strings and Numbers], page 79.

When you change the value of a field (as perceived by `awk`), the text of the input record is recalculated to contain the new field where the old one was. Therefore, `$0` changes to reflect the altered field. Thus, this program prints a copy of the input file, with 10 subtracted from the second field of each line.

```
$ awk '{ $2 = $2 - 10; print $0 }' inventory-shipped
⊣ Jan 3 25 15 115
⊣ Feb 5 32 24 226
⊣ Mar 5 24 34 228
...
```

You can also assign contents to fields that are out of range. For example:

```
$ awk '{ $6 = ($5 + $4 + $3 + $2)
>        print $6 }' inventory-shipped
```

```
⊣ 168
⊣ 297
⊣ 301
...
```

We've just created `$6`, whose value is the sum of fields `$2`, `$3`, `$4`, and `$5`. The '`+`' sign represents addition. For the file '`inventory-shipped`', `$6` represents the total number of parcels shipped for a particular month.

Creating a new field changes `awk`'s internal copy of the current input record—the value of `$0`. Thus, if you do '`print $0`' after adding a field, the record printed includes the new field, with the appropriate number of field separators between it and the previously existing fields.

This recomputation affects and is affected by `NF` (the number of fields; see Section 5.2 [Examining Fields], page 40), and by a feature that has not been discussed yet, the *output field separator*, `OFS`, which is used to separate the fields (see Section 6.3 [Output Separators], page 61). For example, the value of `NF` is set to the number of the highest field you create.

Note, however, that merely *referencing* an out-of-range field does *not* change the value of either `$0` or `NF`. Referencing an out-of-range field only produces an empty string. For example:

```
if ($(NF+1) != "")
    print "can't happen"
else
    print "everything is normal"
```

should print '`everything is normal`', because `NF+1` is certain to be out of range. (See Section 9.1 [The `if-else` Statement], page 103, for more information about `awk`'s `if-else` statements. See Section 7.10 [Variable Typing and Comparison Expressions], page 86, for more information about the '`!=`' operator.)

It is important to note that making an assignment to an existing field will change the value of `$0`, but will not change the value of `NF`, even when you assign the empty string to a field. For example:

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""
>                       print $0; print NF }'
⊣ a::c:d
⊣ 4
```

The field is still there; it just has an empty value. You can tell because there are two colons in a row.

This example shows what happens if you create a new field.

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""; $6 = "new"
>                       print $0; print NF }'
⊣ a::c:d::new
⊣ 6
```

The intervening field, `$5` is created with an empty value (indicated by the second pair of adjacent colons), and `NF` is updated with the value six.

Finally, decrementing `NF` will lose the values of the fields after the new value of `NF`, and `$0` will be recomputed. Here is an example:

```
$ echo a b c d e f | ../gawk '{ print "NF =", NF;
>                               NF = 3; print $0 }'
⊣ NF = 6
⊣ a b c
```

5.5 Specifying How Fields are Separated

This section is rather long; it describes one of the most fundamental operations in `awk`.

5.5.1 The Basics of Field Separating

The *field separator*, which is either a single character or a regular expression, controls the way `awk` splits an input record into fields. `awk` scans the input record for character sequences that match the separator; the fields themselves are the text between the matches.

In the examples below, we use the bullet symbol "•" to represent spaces in the output.

If the field separator is '`oo`', then the following line:

```
moo goo gai pan
```

would be split into three fields: '`m`', '`•g`' and '`•gai•pan`'. Note the leading spaces in the values of the second and third fields.

The field separator is represented by the built-in variable `FS`. Shell programmers take note! `awk` does *not* use the name `IFS` which is used by the POSIX compatible shells (such as the Bourne shell, `sh`, or the GNU Bourne-Again Shell, Bash).

You can change the value of `FS` in the `awk` program with the assignment operator, '`=`' (see Section 7.7 [Assignment Expressions], page 82). Often the right time to do this is at the beginning of execution, before any input has been processed, so that the very first record will be read with the proper separator. To do this, use the special `BEGIN` pattern (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98). For example, here we set the value of `FS` to the string `","`:

```
awk 'BEGIN { FS = "," } ; { print $2 }'
```

Given the input line,

```
John Q. Smith, 29 Oak St., Walamazoo, MI 42139
```

this `awk` program extracts and prints the string '`•29•Oak•St.`'.

Sometimes your input data will contain separator characters that don't separate fields the way you thought they would. For instance, the person's

name in the example we just used might have a title or suffix attached, such as '`John Q. Smith, LXIX`'. From input containing such a name:

```
John Q. Smith, LXIX, 29 Oak St., Walamazoo, MI 42139
```

the above program would extract '`•LXIX`', instead of '`•29•Oak•St.`'. If you were expecting the program to print the address, you would be surprised. The moral is: choose your data layout and separator characters carefully to prevent such problems.

As you know, normally, fields are separated by whitespace sequences (spaces, tabs and newlines), not by single spaces: two spaces in a row do not delimit an empty field. The default value of the field separator `FS` is a string containing a single space, `" "`. If this value were interpreted in the usual way, each space character would separate fields, so two spaces in a row would make an empty field between them. The reason this does not happen is that a single space as the value of `FS` is a special case: it is taken to specify the default manner of delimiting fields.

If `FS` is any other single character, such as `","`, then each occurrence of that character separates two fields. Two consecutive occurrences delimit an empty field. If the character occurs at the beginning or the end of the line, that too delimits an empty field. The space character is the only single character which does not follow these rules.

5.5.2 Using Regular Expressions to Separate Fields

The previous subsection discussed the use of single characters or simple strings as the value of `FS`. More generally, the value of `FS` may be a string containing any regular expression. In this case, each match in the record for the regular expression separates fields. For example, the assignment:

```
FS = ", \t"
```

makes every area of an input line that consists of a comma followed by a space and a tab, into a field separator. ('`\t`' is an *escape sequence* that stands for a tab; see Section 4.2 [Escape Sequences], page 24, for the complete list of similar escape sequences.)

For a less trivial example of a regular expression, suppose you want single spaces to separate fields the way single commas were used above. You can set `FS` to `"[ ]"` (left bracket, space, right bracket). This regular expression matches a single space and nothing else (see Chapter 4 [Regular Expressions], page 23).

There is an important difference between the two cases of '`FS = " "`' (a single space) and '`FS = "[ \t\n]+"`' (left bracket, space, backslash, "t", backslash, "n", right bracket, which is a regular expression matching one or more spaces, tabs, or newlines). For both values of `FS`, fields are separated by runs of spaces, tabs and/or newlines. However, when the value of `FS` is `" "`, `awk` will first strip leading and trailing whitespace from the record, and then decide where the fields are.

For example, the following pipeline prints 'b':

```
$ echo ' a b c d ' | awk '{ print $2 }'
⊣ b
```

However, this pipeline prints 'a' (note the extra spaces around each letter):

```
$ echo ' a  b  c  d ' | awk 'BEGIN { FS = "[ \t]+" }
>                                  { print $2 }'
⊣ a
```

In this case, the first field is *null*, or empty.

The stripping of leading and trailing whitespace also comes into play whenever `$0` is recomputed. For instance, study this pipeline:

```
$ echo '   a b c d' | awk '{ print; $2 = $2; print }'
⊣    a b c d
⊣ a b c d
```

The first `print` statement prints the record as it was read, with leading whitespace intact. The assignment to `$2` rebuilds `$0` by concatenating `$1` through `$NF` together, separated by the value of `OFS`. Since the leading whitespace was ignored when finding `$1`, it is not part of the new `$0`. Finally, the last `print` statement prints the new `$0`.

5.5.3 Making Each Character a Separate Field

There are times when you may want to examine each character of a record separately. In `gawk`, this is easy to do, you simply assign the null string (`""`) to `FS`. In this case, each individual character in the record will become a separate field. Here is an example:

```
$ echo a b | gawk 'BEGIN { FS = "" }
>                 {
>                     for (i = 1; i <= NF; i = i + 1)
>                         print "Field", i, "is", $i
>                 }'
⊣ Field 1 is a
⊣ Field 2 is
⊣ Field 3 is b
```

Traditionally, the behavior for `FS` equal to `""` was not defined. In this case, Unix `awk` would simply treat the entire record as only having one field (d.c.). In compatibility mode (see Section 14.1 [Command Line Options], page 159), if `FS` is the null string, then `gawk` will also behave this way.

5.5.4 Setting `FS` from the Command Line

`FS` can be set on the command line. You use the '`-F`' option to do so. For example:

```
awk -F, 'program' input-files
```

sets `FS` to be the '`,`' character. Notice that the option uses a capital '`F`'. Contrast this with '`-f`', which specifies a file containing an `awk` program. Case is significant in command line options: the '`-F`' and '`-f`' options have nothing to do with each other. You can use both options at the same time to set the `FS` variable *and* get an `awk` program from a file.

The value used for the argument to '`-F`' is processed in exactly the same way as assignments to the built-in variable `FS`. This means that if the field separator contains special characters, they must be escaped appropriately. For example, to use a '`\`' as the field separator, you would have to type:

```
# same as FS = "\\"
awk -F\\\\ '...' files ...
```

Since '`\`' is used for quoting in the shell, `awk` will see '`-F\\`'. Then `awk` processes the '`\\`' for escape characters (see Section 4.2 [Escape Sequences], page 24), finally yielding a single '`\`' to be used for the field separator.

As a special case, in compatibility mode (see Section 14.1 [Command Line Options], page 159), if the argument to '`-F`' is '`t`', then `FS` is set to the tab character. This is because if you type '`-F\t`' at the shell, without any quotes, the '`\`' gets deleted, so `awk` figures that you really want your fields to be separated with tabs, and not '`t`'s. Use '`-v FS="t"`' on the command line if you really do want to separate your fields with '`t`'s (see Section 14.1 [Command Line Options], page 159).

For example, let's use an `awk` program file called '`baud.awk`' that contains the pattern `/300/`, and the action '`print $1`'. Here is the program:

```
/300/   { print $1 }
```

Let's also set `FS` to be the '`-`' character, and run the program on the file '`BBS-list`'. The following command prints a list of the names of the bulletin boards that operate at 300 baud and the first three digits of their phone numbers:

```
$ awk -F- -f baud.awk BBS-list
⊣ aardvark     555
⊣ alpo
⊣ barfly       555
...
```

Note the second line of output. In the original file (see Section 1.3 [Data Files for the Examples], page 7), the second line looked like this:

```
alpo-net     555-3412     2400/1200/300     A
```

The '`-`' as part of the system's name was used as the field separator, instead of the '`-`' in the phone number that was originally intended. This demonstrates why you have to be careful in choosing your field and record separators.

On many Unix systems, each user has a separate entry in the system password file, one line per user. The information in these lines is separated

by colons. The first field is the user's logon name, and the second is the user's encrypted password. A password file entry might look like this:

```
arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh
```

The following program searches the system password file, and prints the entries for users who have no password:

```
awk -F: '$2 == ""' /etc/passwd
```

5.5.5 Field Splitting Summary

According to the POSIX standard, `awk` is supposed to behave as if each record is split into fields at the time that it is read. In particular, this means that you can change the value of `FS` after a record is read, and the value of the fields (i.e. how they were split) should reflect the old value of `FS`, not the new one.

However, many implementations of `awk` do not work this way. Instead, they defer splitting the fields until a field is actually referenced. The fields will be split using the *current* value of `FS`! (d.c.) This behavior can be difficult to diagnose. The following example illustrates the difference between the two methods. (The `sed`[1] command prints just the first line of '`/etc/passwd`'.)

```
sed 1q /etc/passwd | awk '{ FS = ":" ; print $1 }'
```

will usually print

```
root
```

on an incorrect implementation of `awk`, while `gawk` will print something like

```
root:nSijPlPhZZwgE:0:0:Root:/:
```

The following table summarizes how fields are split, based on the value of `FS`. ('`==`' means "is equal to.")

`FS == " "` Fields are separated by runs of whitespace. Leading and trailing whitespace are ignored. This is the default.

`FS ==` *any other single character*
: Fields are separated by each occurrence of the character. Multiple successive occurrences delimit empty fields, as do leading and trailing occurrences. The character can even be a regexp metacharacter; it does not need to be escaped.

`FS ==` *regexp*
: Fields are separated by occurrences of characters that match *regexp*. Leading and trailing matches of *regexp* delimit empty fields.

`FS == ""` Each individual character in the record becomes a separate field.

[1] The `sed` utility is a "stream editor." Its behavior is also defined by the POSIX standard.

5.6 Reading Fixed-width Data

(This section discusses an advanced, experimental feature. If you are a novice `awk` user, you may wish to skip it on the first reading.)

`gawk` version 2.13 introduced a new facility for dealing with fixed-width fields with no distinctive field separator. Data of this nature arises, for example, in the input for old FORTRAN programs where numbers are run together; or in the output of programs that did not anticipate the use of their output as input for other programs.

An example of the latter is a table where all the columns are lined up by the use of a variable number of spaces and *empty fields are just spaces*. Clearly, `awk`'s normal field splitting based on `FS` will not work well in this case. Although a portable `awk` program can use a series of `substr` calls on `$0` (see Section 12.3 [Built-in Functions for String Manipulation], page 135), this is awkward and inefficient for a large number of fields.

The splitting of an input record into fixed-width fields is specified by assigning a string containing space-separated numbers to the built-in variable `FIELDWIDTHS`. Each number specifies the width of the field *including* columns between fields. If you want to ignore the columns between fields, you can specify the width as a separate field that is subsequently ignored.

The following data is the output of the Unix `w` utility. It is useful to illustrate the use of `FIELDWIDTHS`.

```
 10:06pm  up 21 days, 14:04,  23 users
User     tty       login  idle   JCPU   PCPU  what
hzuo     ttyV0     8:58pm            9      5  vi p24.tex
hzang    ttyV3     6:37pm    50                -csh
eklye    ttyV5     9:53pm            7      1  em thes.tex
dportein ttyV6     8:17pm  1:47                -csh
gierd    ttyD3    10:00pm     1                elm
dave     ttyD4     9:47pm            4      4  w
brent    ttyp0    26Jun91  4:46  26:46   4:41  bash
dave     ttyq4    26Jun9115days     46     46  wnewmail
```

The following program takes the above input, converts the idle time to number of seconds and prints out the first two fields and the calculated idle time. (This program uses a number of `awk` features that haven't been introduced yet.)

```
BEGIN  { FIELDWIDTHS = "9 6 10 6 7 7 35" }
NR > 2 {
    idle = $4
    sub(/^  */, "", idle)   # strip leading spaces
    if (idle == "")
        idle = 0
```

```
        if (idle ~ /:/) {
            split(idle, t, ":")
            idle = t[1] * 60 + t[2]
        }
        if (idle ~ /days/)
            idle *= 24 * 60 * 60

        print $1, $2, idle
    }
```

Here is the result of running the program on the data:

```
hzuo      ttyV0  0
hzang     ttyV3  50
eklye     ttyV5  0
dportein  ttyV6  107
gierd     ttyD3  1
dave      ttyD4  0
brent     ttyp0  286
dave      ttyq4  1296000
```

Another (possibly more practical) example of fixed-width input data would be the input from a deck of balloting cards. In some parts of the United States, voters mark their choices by punching holes in computer cards. These cards are then processed to count the votes for any particular candidate or on any particular issue. Since a voter may choose not to vote on some issue, any column on the card may be empty. An `awk` program for processing such data could use the `FIELDWIDTHS` feature to simplify reading the data. (Of course, getting `gawk` to run on a system with card readers is another story!)

Assigning a value to `FS` causes `gawk` to return to using `FS` for field splitting. Use '`FS = FS`' to make this happen, without having to know the current value of `FS`.

This feature is still experimental, and may evolve over time. Note that in particular, `gawk` does not attempt to verify the sanity of the values used in the value of `FIELDWIDTHS`.

5.7 Multiple-Line Records

In some data bases, a single line cannot conveniently hold all the information in one entry. In such cases, you can use multi-line records.

The first step in doing this is to choose your data format: when records are not defined as single lines, how do you want to define them? What should separate records?

One technique is to use an unusual character or string to separate records. For example, you could use the formfeed character (written '`\f`' in `awk`, as in C) to separate them, making each record a page of the file. To do this,

just set the variable `RS` to `"\f"` (a string containing the formfeed character). Any other character could equally well be used, as long as it won't be part of the data in a record.

Another technique is to have blank lines separate records. By a special dispensation, an empty string as the value of `RS` indicates that records are separated by one or more blank lines. If you set `RS` to the empty string, a record always ends at the first blank line encountered. And the next record doesn't start until the first non-blank line that follows—no matter how many blank lines appear in a row, they are considered one record-separator.

You can achieve the same effect as '`RS = ""`' by assigning the string `"\n\n+"` to `RS`. This regexp matches the newline at the end of the record, and one or more blank lines after the record. In addition, a regular expression always matches the longest possible sequence when there is a choice (see Section 4.6 [How Much Text Matches?], page 34). So the next record doesn't start until the first non-blank line that follows—no matter how many blank lines appear in a row, they are considered one record-separator.

There is an important difference between '`RS = ""`' and '`RS = "\n\n+"`'. In the first case, leading newlines in the input data file are ignored, and if a file ends without extra blank lines after the last record, the final newline is removed from the record. In the second case, this special processing is not done (d.c.).

Now that the input is separated into records, the second step is to separate the fields in the record. One way to do this is to divide each of the lines into fields in the normal manner. This happens by default as the result of a special feature: when `RS` is set to the empty string, the newline character *always* acts as a field separator. This is in addition to whatever field separations result from `FS`.

The original motivation for this special exception was probably to provide useful behavior in the default case (i.e. `FS` is equal to `" "`). This feature can be a problem if you really don't want the newline character to separate fields, since there is no way to prevent it. However, you can work around this by using the `split` function to break up the record manually (see Section 12.3 [Built-in Functions for String Manipulation], page 135).

Another way to separate fields is to put each field on a separate line: to do this, just set the variable `FS` to the string `"\n"`. (This simple regular expression matches a single newline.)

A practical example of a data file organized this way might be a mailing list, where each entry is separated by blank lines. If we have a mailing list in a file named '`addresses`', that looks like this:

```
Jane Doe
123 Main Street
Anywhere, SE 12345-6789

John Smith
456 Tree-lined Avenue
Smallville, MW 98765-4321
...
```

A simple program to process this file would look like this:

```
# addrs.awk --- simple mailing list program

# Records are separated by blank lines.
# Each line is one field.
BEGIN { RS = "" ; FS = "\n" }

{
      print "Name is:", $1
      print "Address is:", $2
      print "City and State are:", $3
      print ""
}
```

Running the program produces the following output:

```
$ awk -f addrs.awk addresses
⊣ Name is: Jane Doe
⊣ Address is: 123 Main Street
⊣ City and State are: Anywhere, SE 12345-6789
⊣
⊣ Name is: John Smith
⊣ Address is: 456 Tree-lined Avenue
⊣ City and State are: Smallville, MW 98765-4321
⊣
...
```

See Section 16.2.4 [Printing Mailing Labels], page 228, for a more realistic program that deals with address lists.

The following table summarizes how records are split, based on the value of `RS`. ('`==`' means "is equal to.")

`RS == "\n"`
: Records are separated by the newline character ('`\n`'). In effect, every line in the data file is a separate record, including blank lines. This is the default.

`RS ==` *any single character*
: Records are separated by each occurrence of the character. Multiple successive occurrences delimit empty records.

`RS == ""` Records are separated by runs of blank lines. The newline character always serves as a field separator, in addition to whatever value `FS` may have. Leading and trailing newlines in a file are ignored.

`RS ==` *regexp*
Records are separated by occurrences of characters that match *regexp*. Leading and trailing matches of *regexp* delimit empty records.

In all cases, `gawk` sets `RT` to the input text that matched the value specified by `RS`.

5.8 Explicit Input with `getline`

So far we have been getting our input data from `awk`'s main input stream—either the standard input (usually your terminal, sometimes the output from another program) or from the files specified on the command line. The `awk` language has a special built-in command called `getline` that can be used to read input under your explicit control.

5.8.1 Introduction to `getline`

This command is used in several different ways, and should *not* be used by beginners. It is covered here because this is the chapter on input. The examples that follow the explanation of the `getline` command include material that has not been covered yet. Therefore, come back and study the `getline` command *after* you have reviewed the rest of this book and have a good knowledge of how `awk` works.

`getline` returns one if it finds a record, and zero if the end of the file is encountered. If there is some error in getting a record, such as a file that cannot be opened, then `getline` returns -1. In this case, `gawk` sets the variable `ERRNO` to a string describing the error that occurred.

In the following examples, *command* stands for a string value that represents a shell command.

5.8.2 Using `getline` with No Arguments

The `getline` command can be used without arguments to read input from the current input file. All it does in this case is read the next input record and split it up into fields. This is useful if you've finished processing the current record, but you want to do some special processing *right now* on the next record. Here's an example:

```
awk '{
     if ((t = index($0, "/*")) != 0) {
          # value will be "" if t is 1
          tmp = substr($0, 1, t - 1)
          u = index(substr($0, t + 2), "*/")
          while (u == 0) {
               if (getline <= 0) {
                    m = "unexpected EOF or error"
                    m = (m ": " ERRNO)
                    print m > "/dev/stderr"
                    exit
               }
               t = -1
               u = index($0, "*/")
          }
          # substr expression will be "" if */
          # occurred at end of line
          $0 = tmp substr($0, t + u + 3)
     }
     print $0
}'
```

This `awk` program deletes all C-style comments, '`/* ... */`', from the input. By replacing the '`print $0`' with other statements, you could perform more complicated processing on the decommented input, like searching for matches of a regular expression. This program has a subtle problem—it does not work if one comment ends and another begins on the same line.

This form of the `getline` command sets `NF` (the number of fields; see Section 5.2 [Examining Fields], page 40), `NR` (the number of records read so far; see Section 5.1 [How Input is Split into Records], page 37), `FNR` (the number of records read from this input file), and the value of `$0`.

Note: the new value of `$0` is used in testing the patterns of any subsequent rules. The original value of `$0` that triggered the rule which executed `getline` is lost (d.c.). By contrast, the `next` statement reads a new record but immediately begins processing it normally, starting with the first rule in the program. See Section 9.7 [The `next` Statement], page 108.

5.8.3 Using `getline` Into a Variable

You can use '`getline` *var*' to read the next record from `awk`'s input into the variable *var*. No other processing is done.

For example, suppose the next line is a comment, or a special string, and you want to read it, without triggering any rules. This form of `getline` allows you to read that line and store it in a variable so that the main read-a-line-and-check-each-rule loop of `awk` never sees it.

The following example swaps every two lines of input. For example, given:

```
wan
tew
free
phore
```

it outputs:

```
tew
wan
phore
free
```

Here's the program:

```
awk '{
     if ((getline tmp) > 0) {
          print tmp
          print $0
     } else
          print $0
}'
```

The `getline` command used in this way sets only the variables `NR` and `FNR` (and of course, *var*). The record is not split into fields, so the values of the fields (including `$0`) and the value of `NF` do not change.

5.8.4 Using `getline` from a File

Use '`getline <` *file*' to read the next record from the file *file*. Here *file* is a string-valued expression that specifies the file name. '`<` *file*' is called a *redirection* since it directs input to come from a different place.

For example, the following program reads its input record from the file '`secondary.input`' when it encounters a first field with a value equal to 10 in the current input file.

```
awk '{
    if ($1 == 10) {
         getline < "secondary.input"
         print
    } else
         print
}'
```

Since the main input stream is not used, the values of `NR` and `FNR` are not changed. But the record read is split into fields in the normal manner, so the values of `$0` and other fields are changed. So is the value of `NF`.

According to POSIX, '`getline <` *expression*' is ambiguous if *expression* contains unparenthesized operators other than '`$`'; for example, '`getline < dir "/" file`' is ambiguous because the concatenation operator is not

parenthesized, and you should write it as '`getline < (dir "/" file)`' if you want your program to be portable to other `awk` implementations.

5.8.5 Using `getline` Into a Variable from a File

Use '`getline` *var* `<` *file*' to read input the file *file* and put it in the variable *var*. As above, *file* is a string-valued expression that specifies the file from which to read.

In this version of `getline`, none of the built-in variables are changed, and the record is not split into fields. The only variable changed is *var*.

For example, the following program copies all the input files to the output, except for records that say '`@include` *filename*'. Such a record is replaced by the contents of the file *filename*.

```
awk '{
     if (NF == 2 && $1 == "@include") {
          while ((getline line < $2) > 0)
               print line
          close($2)
     } else
          print
}'
```

Note here how the name of the extra input file is not built into the program; it is taken directly from the data, from the second field on the '`@include`' line.

The `close` function is called to ensure that if two identical '`@include`' lines appear in the input, the entire specified file is included twice. See Section 6.8 [Closing Input and Output Files and Pipes], page 71.

One deficiency of this program is that it does not process nested '`@include`' statements ('`@include`' statements in included files) the way a true macro preprocessor would. See Section 16.2.9 [An Easy Way to Use Library Functions], page 239, for a program that does handle nested '`@include`' statements.

5.8.6 Using `getline` from a Pipe

You can pipe the output of a command into `getline`, using '*command* `| getline`'. In this case, the string *command* is run as a shell command and its output is piped into `awk` to be used as input. This form of `getline` reads one record at a time from the pipe.

For example, the following program copies its input to its output, except for lines that begin with '`@execute`', which are replaced by the output produced by running the rest of the line as a shell command:

```
awk '{
     if ($1 == "@execute") {
          tmp = substr($0, 10)
          while ((tmp | getline) > 0)
               print
          close(tmp)
     } else
          print
}'
```

The `close` function is called to ensure that if two identical '`@execute`' lines appear in the input, the command is run for each one. See Section 6.8 [Closing Input and Output Files and Pipes], page 71.

Given the input:

```
foo
bar
baz
@execute who
bletch
```

the program might produce:

```
foo
bar
baz
arnold     ttyv0   Jul 13 14:22
miriam     ttyp0   Jul 13 14:23     (murphy:0)
bill       ttyp1   Jul 13 14:23     (murphy:0)
bletch
```

Notice that this program ran the command `who` and printed the result. (If you try this program yourself, you will of course get different results, showing you who is logged in on your system.)

This variation of `getline` splits the record into fields, sets the value of `NF` and recomputes the value of `$0`. The values of `NR` and `FNR` are not changed.

According to POSIX, '*expression* `| getline`' is ambiguous if *expression* contains unparenthesized operators other than '`$`'; for example, '`"echo " "date" | getline`' is ambiguous because the concatenation operator is not parenthesized, and you should write it as '`("echo " "date") | getline`' if you want your program to be portable to other `awk` implementations. (It happens that `gawk` gets it right, but you should not rely on this. Parentheses make it easier to read, anyway.)

5.8.7 Using `getline` Into a Variable from a Pipe

When you use '*command* `| getline` *var*', the output of the command *command* is sent through a pipe to `getline` and into the variable *var*. For

example, the following program reads the current date and time into the variable `current_time`, using the `date` utility, and then prints it.

```
awk 'BEGIN {
     "date" | getline current_time
     close("date")
     print "Report printed on " current_time
}'
```

In this version of `getline`, none of the built-in variables are changed, and the record is not split into fields.

5.8.8 Summary of `getline` Variants

With all the forms of `getline`, even though `$0` and `NF`, may be updated, the record will not be tested against all the patterns in the `awk` program, in the way that would happen if the record were read normally by the main processing loop of `awk`. However the new record is tested against any subsequent rules.

Many `awk` implementations limit the number of pipelines an `awk` program may have open to just one! In `gawk`, there is no such limit. You can open as many pipelines as the underlying operating system will permit.

An interesting side-effect occurs if you use `getline` (without a redirection) inside a `BEGIN` rule. Since an unredirected `getline` reads from the command line data files, the first `getline` command causes `awk` to set the value of `FILENAME`. Normally, `FILENAME` does not have a value inside `BEGIN` rules, since you have not yet started to process the command line data files (d.c.). (See Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98, also see Section 10.2 [Built-in Variables that Convey Information], page 115.)

The following table summarizes the six variants of `getline`, listing which built-in variables are set by each one.

`getline`
: sets `$0`, `NF`, `FNR`, and `NR`.

`getline` *var*
: sets *var*, `FNR`, and `NR`.

`getline <` *file*
: sets `$0`, and `NF`.

`getline` *var* `<` *file*
: sets *var*.

command `| getline`
: sets `$0`, and `NF`.

command `| getline` *var*
: sets *var*.

6 Printing Output

One of the most common actions is to *print*, or output, some or all of the input. You use the `print` statement for simple output. You use the `printf` statement for fancier formatting. Both are described in this chapter.

6.1 The `print` Statement

The `print` statement does output with simple, standardized formatting. You specify only the strings or numbers to be printed, in a list separated by commas. They are output, separated by single spaces, followed by a newline. The statement looks like this:

`print` *item1*`,` *item2*`,` ...

The entire list of items may optionally be enclosed in parentheses. The parentheses are necessary if any of the item expressions uses the '`>`' relational operator; otherwise it could be confused with a redirection (see Section 6.6 [Redirecting Output of `print` and `printf`], page 67).

The items to be printed can be constant strings or numbers, fields of the current record (such as `$1`), variables, or any `awk` expressions. Numeric values are converted to strings, and then printed.

The `print` statement is completely general for computing *what* values to print. However, with two exceptions, you cannot specify *how* to print them—how many columns, whether to use exponential notation or not, and so on. (For the exceptions, see Section 6.3 [Output Separators], page 61, and Section 6.4 [Controlling Numeric Output with `print`], page 62.) For that, you need the `printf` statement (see Section 6.5 [Using `printf` Statements for Fancier Printing], page 62).

The simple statement '`print`' with no items is equivalent to '`print $0`': it prints the entire current record. To print a blank line, use '`print ""`', where `""` is the empty string.

To print a fixed piece of text, use a string constant such as `"Don't Panic"` as one item. If you forget to use the double-quote characters, your text will be taken as an `awk` expression, and you will probably get an error. Keep in mind that a space is printed between any two items.

Each `print` statement makes at least one line of output. But it isn't limited to one line. If an item value is a string that contains a newline, the newline is output along with the rest of the string. A single `print` can make any number of lines this way.

6.2 Examples of `print` Statements

Here is an example of printing a string that contains embedded newlines (the '`\n`' is an escape sequence, used to represent the newline character; see Section 4.2 [Escape Sequences], page 24):

```
$ awk 'BEGIN { print "line one\nline two\nline three" }'
⊣ line one
⊣ line two
⊣ line three
```

Here is an example that prints the first two fields of each input record, with a space between them:

```
$ awk '{ print $1, $2 }' inventory-shipped
⊣ Jan 13
⊣ Feb 15
⊣ Mar 15
...
```

A common mistake in using the `print` statement is to omit the comma between two items. This often has the effect of making the items run together in the output, with no space. The reason for this is that juxtaposing two string expressions in `awk` means to concatenate them. Here is the same program, without the comma:

```
$ awk '{ print $1 $2 }' inventory-shipped
⊣ Jan13
⊣ Feb15
⊣ Mar15
...
```

To someone unfamiliar with the file '`inventory-shipped`', neither example's output makes much sense. A heading line at the beginning would make it clearer. Let's add some headings to our table of months (`$1`) and green crates shipped (`$2`). We do this using the `BEGIN` pattern (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98) to force the headings to be printed only once:

```
awk 'BEGIN {  print "Month Crates"
              print "-----  ------" }
           {  print $1, $2 }' inventory-shipped
```

Did you already guess what happens? When run, the program prints the following:

```
Month Crates
-----  ------
Jan 13
Feb 15
Mar 15
...
```

The headings and the table data don't line up! We can fix this by printing some spaces between the two fields:

```
awk 'BEGIN { print "Month Crates"
             print "-----  ------" }
           { print $1, "     ", $2 }' inventory-shipped
```

You can imagine that this way of lining up columns can get pretty complicated when you have many columns to fix. Counting spaces for two or three columns can be simple, but more than this and you can get lost quite easily. This is why the `printf` statement was created (see Section 6.5 [Using `printf` Statements for Fancier Printing], page 62); one of its specialties is lining up columns of data.

As a side point, you can continue either a `print` or `printf` statement simply by putting a newline after any comma (see Section 2.6 [`awk` Statements Versus Lines], page 16).

6.3 Output Separators

As mentioned previously, a `print` statement contains a list of items, separated by commas. In the output, the items are normally separated by single spaces. This need not be the case; a single space is only the default. You can specify any string of characters to use as the *output field separator* by setting the built-in variable `OFS`. The initial value of this variable is the string `" "`, that is, a single space.

The output from an entire `print` statement is called an *output record*. Each `print` statement outputs one output record and then outputs a string called the *output record separator*. The built-in variable `ORS` specifies this string. The initial value of `ORS` is the string `"\n"`, i.e. a newline character; thus, normally each `print` statement makes a separate line.

You can change how output fields and records are separated by assigning new values to the variables `OFS` and/or `ORS`. The usual place to do this is in the `BEGIN` rule (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98), so that it happens before any input is processed. You may also do this with assignments on the command line, before the names of your input files, or using the '`-v`' command line option (see Section 14.1 [Command Line Options], page 159).

The following example prints the first and second fields of each input record separated by a semicolon, with a blank line added after each line:

```
$ awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
>            { print $1, $2 }' BBS-list
 ⊣ aardvark;555-5553
 ⊣
 ⊣ alpo-net;555-3412
 ⊣
 ⊣ barfly;555-7685
...
```

If the value of `ORS` does not contain a newline, all your output will be run together on a single line, unless you output newlines some other way.

6.4 Controlling Numeric Output with `print`

When you use the `print` statement to print numeric values, `awk` internally converts the number to a string of characters, and prints that string. `awk` uses the `sprintf` function to do this conversion (see Section 12.3 [Built-in Functions for String Manipulation], page 135). For now, it suffices to say that the `sprintf` function accepts a *format specification* that tells it how to format numbers (or strings), and that there are a number of different ways in which numbers can be formatted. The different format specifications are discussed more fully in Section 6.5.2 [Format-Control Letters], page 63.

The built-in variable `OFMT` contains the default format specification that `print` uses with `sprintf` when it wants to convert a number to a string for printing. The default value of `OFMT` is `"%.6g"`. By supplying different format specifications as the value of `OFMT`, you can change how `print` will print your numbers. As a brief example:

```
$ awk 'BEGIN {
>   OFMT = "%.0f"  # print numbers as integers (rounds)
>   print 17.23 }'
⊣ 17
```

According to the POSIX standard, `awk`'s behavior will be undefined if `OFMT` contains anything but a floating point conversion specification (d.c.).

6.5 Using `printf` Statements for Fancier Printing

If you want more precise control over the output format than `print` gives you, use `printf`. With `printf` you can specify the width to use for each item, and you can specify various formatting choices for numbers (such as what radix to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point). You do this by supplying a string, called the *format string*, which controls how and where to print the other arguments.

6.5.1 Introduction to the `printf` Statement

The `printf` statement looks like this:

`printf` *format*`,` *item1*`,` *item2*`, ...`

The entire list of arguments may optionally be enclosed in parentheses. The parentheses are necessary if any of the item expressions use the '`>`' relational operator; otherwise it could be confused with a redirection (see Section 6.6 [Redirecting Output of `print` and `printf`], page 67).

The difference between `printf` and `print` is the *format* argument. This is an expression whose value is taken as a string; it specifies how to output each of the other arguments. It is called the *format string*.

The format string is very similar to that in the ANSI C library function `printf`. Most of *format* is text to be output verbatim. Scattered among this text are *format specifiers*, one per item. Each format specifier says to output the next item in the argument list at that place in the format.

The `printf` statement does not automatically append a newline to its output. It outputs only what the format string specifies. So if you want a newline, you must include one in the format string. The output separator variables `OFS` and `ORS` have no effect on `printf` statements. For example:

```
BEGIN {
   ORS = "\nOUCH!\n"; OFS = "!"
   msg = "Don't Panic!"; printf "%s\n", msg
}
```

This program still prints the familiar '`Don't Panic!`' message.

6.5.2 Format-Control Letters

A format specifier starts with the character '`%`' and ends with a *format-control letter*; it tells the `printf` statement how to output one item. (If you actually want to output a '`%`', write '`%%`'.) The format-control letter specifies what kind of value to print. The rest of the format specifier is made up of optional *modifiers* which are parameters to use, such as the field width.

Here is a list of the format-control letters:

`c` This prints a number as an ASCII character. Thus, '`printf "%c", 65`' outputs the letter '`A`'. The output for a string value is the first character of the string.

`d`
`i` These are equivalent. They both print a decimal integer. The '`%i`' specification is for compatibility with ANSI C.

`e`
`E` This prints a number in scientific (exponential) notation. For example,

```
printf "%4.3e\n", 1950
```

prints '`1.950e+03`', with a total of four significant figures of which three follow the decimal point. The '`4.3`' are modifiers, discussed below. '`%E`' uses '`E`' instead of '`e`' in the output.

`f` This prints a number in floating point notation. For example,

```
printf "%4.3f", 1950
```

prints '`1950.000`', with a total of four significant figures of which three follow the decimal point. The '`4.3`' are modifiers, discussed below.

`g`
`G` This prints a number in either scientific notation or floating point notation, whichever uses fewer characters. If the result is printed in scientific notation, '`%G`' uses '`E`' instead of '`e`'.

`o` This prints an unsigned octal integer. (In octal, or base-eight notation, the digits run from '`0`' to '`7`'; the decimal number eight is represented as '`10`' in octal.)

`s` This prints a string.

`u` This prints an unsigned decimal number. (This format is of marginal use, since all numbers in `awk` are floating point. It is provided primarily for compatibility with C.)

`x`
`X` This prints an unsigned hexadecimal integer. (In hexadecimal, or base-16 notation, the digits are '`0`' through '`9`' and '`a`' through '`f`'. The hexadecimal digit '`f`' represents the decimal number 15.) '`%X`' uses the letters '`A`' through '`F`' instead of '`a`' through '`f`'.

`%` This isn't really a format-control letter, but it does have a meaning when used after a '`%`': the sequence '`%%`' outputs one '`%`'. It does not consume an argument, and it ignores any modifiers.

When using the integer format-control letters for values that are outside the range of a C `long` integer, `gawk` will switch to the '`%g`' format specifier. Other versions of `awk` may print invalid values, or do something else entirely (d.c.).

6.5.3 Modifiers for `printf` Formats

A format specification can also include *modifiers* that can control how much of the item's value is printed and how much space it gets. The modifiers come between the '`%`' and the format-control letter. In the examples below, we use the bullet symbol "•" to represent spaces in the output. Here are the possible modifiers, in the order in which they may appear:

`-` The minus sign, used before the width modifier (see below), says to left-justify the argument within its specified width. Normally the argument is printed right-justified in the specified width. Thus,

```
printf "%-4s", "foo"
```

prints '`foo•`'.

space For numeric conversions, prefix positive values with a space, and negative values with a minus sign.

`+` The plus sign, used before the width modifier (see below), says to always supply a sign for numeric conversions, even if the data to be formatted is positive. The '`+`' overrides the space modifier.

`#` Use an "alternate form" for certain control letters. For '`%o`', supply a leading zero. For '`%x`', and '`%X`', supply a leading '`0x`' or '`0X`' for a non-zero result. For '`%e`', '`%E`', and '`%f`', the result will always contain a decimal point. For '`%g`', and '`%G`', trailing zeros are not removed from the result.

`0` A leading '`0`' (zero) acts as a flag, that indicates output should be padded with zeros instead of spaces. This applies even to non-numeric output formats (d.c.). This flag only has an effect when the field width is wider than the value to be printed.

width This is a number specifying the desired minimum width of a field. Inserting any number between the '`%`' sign and the format control character forces the field to be expanded to this width. The default way to do this is to pad with spaces on the left. For example,

```
printf "%4s", "foo"
```

prints '`•foo`'.

The value of *width* is a minimum width, not a maximum. If the item value requires more than *width* characters, it can be as wide as necessary. Thus,

```
printf "%4s", "foobar"
```

prints '`foobar`'.

Preceding the *width* with a minus sign causes the output to be padded with spaces on the right, instead of on the left.

.prec This is a number that specifies the precision to use when printing. For the '`e`', '`E`', and '`f`' formats, this specifies the number of digits you want printed to the right of the decimal point. For the '`g`', and '`G`' formats, it specifies the maximum number of significant digits. For the '`d`', '`o`', '`i`', '`u`', '`x`', and '`X`' formats, it specifies the minimum number of digits to print. For a string, it specifies the maximum number of characters from the string that should be printed. Thus,

```
printf "%.4s", "foobar"
```

prints '`foob`'.

The C library `printf`'s dynamic *width* and *prec* capability (for example, "`%*.*s`") is supported. Instead of supplying explicit *width* and/or *prec* values in the format string, you pass them in the argument list. For example:

```
w = 5
p = 3
```

```
s = "abcdefg"
printf "%*.*s\n", w, p, s
```

is exactly equivalent to

```
s = "abcdefg"
printf "%5.3s\n", s
```

Both programs output '••abc'.

Earlier versions of `awk` did not support this capability. If you must use such a version, you may simulate this feature by using concatenation to build up the format string, like so:

```
w = 5
p = 3
s = "abcdefg"
printf "%" w "." p "s\n", s
```

This is not particularly easy to read, but it does work.

C programmers may be used to supplying additional '`l`' and '`h`' flags in `printf` format strings. These are not valid in `awk`. Most `awk` implementations silently ignore these flags. If '`--lint`' is provided on the command line (see Section 14.1 [Command Line Options], page 159), `gawk` will warn about their use. If '`--posix`' is supplied, their use is a fatal error.

6.5.4 Examples Using `printf`

Here is how to use `printf` to make an aligned table:

```
awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
```

prints the names of bulletin boards (`$1`) of the file '`BBS-list`' as a string of 10 characters, left justified. It also prints the phone numbers (`$2`) afterward on the line. This produces an aligned two-column table of names and phone numbers:

```
$ awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
⊣ aardvark   555-5553
⊣ alpo-net   555-3412
⊣ barfly     555-7685
⊣ bites      555-1675
⊣ camelot    555-0542
⊣ core       555-2912
⊣ fooey      555-1234
⊣ foot       555-6699
⊣ macfoo     555-6480
⊣ sdace      555-3430
⊣ sabafoo    555-2127
```

Did you notice that we did not specify that the phone numbers be printed as numbers? They had to be printed as strings because the numbers are separated by a dash. If we had tried to print the phone numbers as numbers,

all we would have gotten would have been the first three digits, '555'. This would have been pretty confusing.

We did not specify a width for the phone numbers because they are the last things on their lines. We don't need to put spaces after them.

We could make our table look even nicer by adding headings to the tops of the columns. To do this, we use the `BEGIN` pattern (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98) to force the header to be printed only once, at the beginning of the `awk` program:

```
awk 'BEGIN { print "Name      Number"
             print "----      ------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

Did you notice that we mixed `print` and `printf` statements in the above example? We could have used just `printf` statements to get the same results:

```
awk 'BEGIN { printf "%-10s %s\n", "Name", "Number"
             printf "%-10s %s\n", "----", "------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

By printing each column heading with the same format specification used for the elements of the column, we have made sure that the headings are aligned just like the columns.

The fact that the same format specification is used three times can be emphasized by storing it in a variable, like this:

```
awk 'BEGIN { format = "%-10s %s\n"
             printf format, "Name", "Number"
             printf format, "----", "------" }
     { printf format, $1, $2 }' BBS-list
```

See if you can use the `printf` statement to line up the headings and table data for our '`inventory-shipped`' example covered earlier in the section on the `print` statement (see Section 6.1 [The `print` Statement], page 59).

6.6 Redirecting Output of `print` and `printf`

So far we have been dealing only with output that prints to the standard output, usually your terminal. Both `print` and `printf` can also send their output to other places. This is called *redirection*.

A redirection appears after the `print` or `printf` statement. Redirections in `awk` are written just like redirections in shell commands, except that they are written inside the `awk` program.

There are three forms of output redirection: output to a file, output appended to a file, and output through a pipe to another command. They are all shown for the `print` statement, but they work identically for `printf` also.

`print` *items* `>` *output-file*
: This type of redirection prints the items into the output file *output-file*. The file name *output-file* can be any expression. Its value is changed to a string and then used as a file name (see Chapter 7 [Expressions], page 75).

  When this type of redirection is used, the *output-file* is erased before the first output is written to it. Subsequent writes to the same *output-file* do not erase *output-file*, but append to it. If *output-file* does not exist, then it is created.

  For example, here is how an `awk` program can write a list of BBS names to a file '`name-list`' and a list of phone numbers to a file '`phone-list`'. Each output file contains one name or number per line.

```
$ awk '{ print $2 > "phone-list"
>        print $1 > "name-list" }' BBS-list
$ cat phone-list
⊣ 555-5553
⊣ 555-3412
...
$ cat name-list
⊣ aardvark
⊣ alpo-net
...
```

`print` *items* `>>` *output-file*
: This type of redirection prints the items into the pre-existing output file *output-file*. The difference between this and the single-'`>`' redirection is that the old contents (if any) of *output-file* are not erased. Instead, the `awk` output is appended to the file. If *output-file* does not exist, then it is created.

`print` *items* `|` *command*
: It is also possible to send output to another program through a pipe instead of into a file. This type of redirection opens a pipe to *command* and writes the values of *items* through this pipe, to another process created to execute *command*.

  The redirection argument *command* is actually an `awk` expression. Its value is converted to a string, whose contents give the shell command to be run.

  For example, this produces two files, one unsorted list of BBS names and one list sorted in reverse alphabetical order:

```
awk '{ print $1 > "names.unsorted"
       command = "sort -r > names.sorted"
       print $1 | command }' BBS-list
```

Here the unsorted list is written with an ordinary redirection while the sorted list is written by piping through the `sort` utility.

This example uses redirection to mail a message to a mailing list '`bug-system`'. This might be useful when trouble is encountered in an `awk` script run periodically for system maintenance.

```
report = "mail bug-system"
print "Awk script failed:", $0 | report
m = ("at record number " FNR " of " FILENAME)
print m | report
close(report)
```

The message is built using string concatenation and saved in the variable `m`. It is then sent down the pipeline to the `mail` program.

We call the `close` function here because it's a good idea to close the pipe as soon as all the intended output has been sent to it. See Section 6.8 [Closing Input and Output Files and Pipes], page 71, for more information on this. This example also illustrates the use of a variable to represent a *file* or *command*: it is not necessary to always use a string constant. Using a variable is generally a good idea, since `awk` requires you to spell the string value identically every time.

Redirecting output using '`>`', '`>>`', or '`|`' asks the system to open a file or pipe only if the particular *file* or *command* you've specified has not already been written to by your program, or if it has been closed since it was last written to.

As mentioned earlier (see Section 5.8.8 [Summary of `getline` Variants], page 58), many `awk` implementations limit the number of pipelines an `awk` program may have open to just one! In `gawk`, there is no such limit. You can open as many pipelines as the underlying operating system will permit.

6.7 Special File Names in `gawk`

Running programs conventionally have three input and output streams already available to them for reading and writing. These are known as the *standard input*, *standard output*, and *standard error output*. These streams are, by default, connected to your terminal, but they are often redirected with the shell, via the '`<`', '`<<`', '`>`', '`>>`', '`>&`' and '`|`' operators. Standard error is typically used for writing error messages; the reason we have two separate streams, standard output and standard error, is so that they can be redirected separately.

In other implementations of `awk`, the only way to write an error message to standard error in an `awk` program is as follows:

```
print "Serious error detected!" | "cat 1>&2"
```

This works by opening a pipeline to a shell command which can access the standard error stream which it inherits from the `awk` process. This is far from elegant, and is also inefficient, since it requires a separate process. So people writing `awk` programs often neglect to do this. Instead, they send the error messages to the terminal, like this:

```
print "Serious error detected!" > "/dev/tty"
```

This usually has the same effect, but not always: although the standard error stream is usually the terminal, it can be redirected, and when that happens, writing to the terminal is not correct. In fact, if `awk` is run from a background job, it may not have a terminal at all. Then opening '`/dev/tty`' will fail.

`gawk` provides special file names for accessing the three standard streams. When you redirect input or output in `gawk`, if the file name matches one of these special names, then `gawk` directly uses the stream it stands for.

'`/dev/stdin`'
: The standard input (file descriptor 0).

'`/dev/stdout`'
: The standard output (file descriptor 1).

'`/dev/stderr`'
: The standard error output (file descriptor 2).

'`/dev/fd/`*N*'
: The file associated with file descriptor *N*. Such a file must have been opened by the program initiating the `awk` execution (typically the shell). Unless you take special pains in the shell from which you invoke `gawk`, only descriptors 0, 1 and 2 are available.

The file names '`/dev/stdin`', '`/dev/stdout`', and '`/dev/stderr`' are aliases for '`/dev/fd/0`', '`/dev/fd/1`', and '`/dev/fd/2`', respectively, but they are more self-explanatory.

The proper way to write an error message in a `gawk` program is to use '`/dev/stderr`', like this:

```
print "Serious error detected!" > "/dev/stderr"
```

`gawk` also provides special file names that give access to information about the running `gawk` process. Each of these "files" provides a single record of information. To read them more than once, you must first close them with the `close` function (see Section 6.8 [Closing Input and Output Files and Pipes], page 71). The filenames are:

'`/dev/pid`'
: Reading this file returns the process ID of the current process, in decimal, terminated with a newline.

'`/dev/ppid`'
: Reading this file returns the parent process ID of the current process, in decimal, terminated with a newline.

'`/dev/pgrpid`'
: Reading this file returns the process group ID of the current process, in decimal, terminated with a newline.

'`/dev/user`'
: Reading this file returns a single record terminated with a newline. The fields are separated with spaces. The fields represent the following information:

  `$1` The return value of the `getuid` system call (the real user ID number).

  `$2` The return value of the `geteuid` system call (the effective user ID number).

  `$3` The return value of the `getgid` system call (the real group ID number).

  `$4` The return value of the `getegid` system call (the effective group ID number).

  If there are any additional fields, they are the group IDs returned by `getgroups` system call. (Multiple groups may not be supported on all systems.)

These special file names may be used on the command line as data files, as well as for I/O redirections within an `awk` program. They may not be used as source files with the '`-f`' option.

Recognition of these special file names is disabled if `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159).

Caution: Unless your system actually has a '`/dev/fd`' directory (or any of the other above listed special files), the interpretation of these file names is done by `gawk` itself. For example, using '`/dev/fd/4`' for output will actually write on file descriptor 4, and not on a new file descriptor that was `dup`'ed from file descriptor 4. Most of the time this does not matter; however, it is important to *not* close any of the files related to file descriptors 0, 1, and 2. If you do close one of these files, unpredictable behavior will result.

The special files that provide process-related information will disappear in a future version of `gawk`. See Section C.3 [Probable Future Extensions], page 293.

6.8 Closing Input and Output Files and Pipes

If the same file name or the same shell command is used with `getline` (see Section 5.8 [Explicit Input with `getline`], page 53) more than once during the execution of an `awk` program, the file is opened (or the command is executed) only the first time. At that time, the first record of input is read from that file or command. The next time the same file or command is used in `getline`, another record is read from it, and so on.

Similarly, when a file or pipe is opened for output, the file name or command associated with it is remembered by `awk` and subsequent writes to the same file or command are appended to the previous writes. The file or pipe stays open until `awk` exits.

This implies that if you want to start reading the same file again from the beginning, or if you want to rerun a shell command (rather than reading more output from the command), you must take special steps. What you must do is use the `close` function, as follows:

```
close(filename)
```

or

```
close(command)
```

The argument *filename* or *command* can be any expression. Its value must *exactly* match the string that was used to open the file or start the command (spaces and other "irrelevant" characters included). For example, if you open a pipe with this:

```
"sort -r names" | getline foo
```

then you must close it with this:

```
close("sort -r names")
```

Once this function call is executed, the next `getline` from that file or command, or the next `print` or `printf` to that file or command, will reopen the file or rerun the command.

Because the expression that you use to close a file or pipeline must exactly match the expression used to open the file or run the command, it is good practice to use a variable to store the file name or command. The previous example would become

```
sortcom = "sort -r names"
sortcom | getline foo
...
close(sortcom)
```

This helps avoid hard-to-find typographical errors in your `awk` programs.

Here are some reasons why you might need to close an output file:

- To write a file and read it back later on in the same `awk` program. Close the file when you are finished writing it; then you can start reading it with `getline`.
- To write numerous files, successively, in the same `awk` program. If you don't close the files, eventually you may exceed a system limit on the number of open files in one process. So close each one when you are finished writing it.
- To make a command finish. When you redirect output through a pipe, the command reading the pipe normally continues to try to read input as long as the pipe is open. Often this means the command cannot really do its work until the pipe is closed. For example, if you redirect

output to the `mail` program, the message is not actually sent until the pipe is closed.

- To run the same program a second time, with the same arguments. This is not the same thing as giving more input to the first run!

  For example, suppose you pipe output to the `mail` program. If you output several lines redirected to this pipe without closing it, they make a single message of several lines. By contrast, if you close the pipe after each line of output, then each line makes a separate message.

`close` returns a value of zero if the close succeeded. Otherwise, the value will be non-zero. In this case, `gawk` sets the variable `ERRNO` to a string describing the error that occurred.

If you use more files than the system allows you to have open, `gawk` will attempt to multiplex the available open files among your data files. `gawk`'s ability to do this depends upon the facilities of your operating system: it may not always work. It is therefore both good practice and good portability advice to always use `close` on your files when you are done with them.

7 Expressions

Expressions are the basic building blocks of `awk` patterns and actions. An expression evaluates to a value, which you can print, test, store in a variable or pass to a function. Additionally, an expression can assign a new value to a variable or a field, with an assignment operator.

An expression can serve as a pattern or action statement on its own. Most other kinds of statements contain one or more expressions which specify data on which to operate. As in other languages, expressions in `awk` include variables, array references, constants, and function calls, as well as combinations of these with various operators.

7.1 Constant Expressions

The simplest type of expression is the *constant*, which always has the same value. There are three types of constants: numeric constants, string constants, and regular expression constants.

7.1.1 Numeric and String Constants

A *numeric constant* stands for a number. This number can be an integer, a decimal fraction, or a number in scientific (exponential) notation.[1] Here are some examples of numeric constants, which all have the same value:

```
105
1.05e+2
1050e-1
```

A string constant consists of a sequence of characters enclosed in double-quote marks. For example:

```
"parrot"
```

represents the string whose contents are '`parrot`'. Strings in `gawk` can be of any length and they can contain any of the possible eight-bit ASCII characters including ASCII NUL (character code zero). Other `awk` implementations may have difficulty with some character codes.

7.1.2 Regular Expression Constants

A regexp constant is a regular expression description enclosed in slashes, such as `/^beginning and end$/`. Most regexps used in `awk` programs are constant, but the '`~`' and '`!~`' matching operators can also match computed or "dynamic" regexps (which are just ordinary strings or variables that contain a regexp).

---

[1] The internal representation uses double-precision floating point numbers. If you don't know what that means, then don't worry about it.

7.2 Using Regular Expression Constants

When used on the right hand side of the '`~`' or '`!~`' operators, a regexp constant merely stands for the regexp that is to be matched.

Regexp constants (such as `/foo/`) may be used like simple expressions. When a regexp constant appears by itself, it has the same meaning as if it appeared in a pattern, i.e. '`($0 ~ /foo/)`' (d.c.) (see Section 8.1.3 [Expressions as Patterns], page 96). This means that the two code segments,

```
if ($0 ~ /barfly/ || $0 ~ /camelot/)
    print "found"
```

and

```
if (/barfly/ || /camelot/)
    print "found"
```

are exactly equivalent.

One rather bizarre consequence of this rule is that the following boolean expression is valid, but does not do what the user probably intended:

```
# note that /foo/ is on the left of the ~
if (/foo/ ~ $1) print "found foo"
```

This code is "obviously" testing `$1` for a match against the regexp `/foo/`. But in fact, the expression '`/foo/ ~ $1`' actually means '`($0 ~ /foo/) ~ $1`'. In other words, first match the input record against the regexp `/foo/`. The result will be either zero or one, depending upon the success or failure of the match. Then match that result against the first field in the record.

Since it is unlikely that you would ever really wish to make this kind of test, `gawk` will issue a warning when it sees this construct in a program.

Another consequence of this rule is that the assignment statement

```
matches = /foo/
```

will assign either zero or one to the variable `matches`, depending upon the contents of the current input record.

This feature of the language was never well documented until the POSIX specification.

Constant regular expressions are also used as the first argument for the `gensub`, `sub` and `gsub` functions, and as the second argument of the `match` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135). Modern implementations of `awk`, including `gawk`, allow the third argument of `split` to be a regexp constant, while some older implementations do not (d.c.).

This can lead to confusion when attempting to use regexp constants as arguments to user defined functions (see Chapter 13 [User-defined Functions], page 151). For example:

```
function mysub(pat, repl, str, global)
{
    if (global)
        gsub(pat, repl, str)
    else
        sub(pat, repl, str)
    return str
}

{
    ...
    text = "hi! hi yourself!"
    mysub(/hi/, "howdy", text, 1)
    ...
}
```

In this example, the programmer wishes to pass a regexp constant to the user-defined function `mysub`, which will in turn pass it on to either `sub` or `gsub`. However, what really happens is that the `pat` parameter will be either one or zero, depending upon whether or not `$0` matches `/hi/`.

As it is unlikely that you would ever really wish to pass a truth value in this way, `gawk` will issue a warning when it sees a regexp constant used as a parameter to a user-defined function.

7.3 Variables

Variables are ways of storing values at one point in your program for use later in another part of your program. You can manipulate them entirely within your program text, and you can also assign values to them on the `awk` command line.

7.3.1 Using Variables in a Program

Variables let you give names to values and refer to them later. You have already seen variables in many of the examples. The name of a variable must be a sequence of letters, digits and underscores, but it may not begin with a digit. Case is significant in variable names; `a` and `A` are distinct variables.

A variable name is a valid expression by itself; it represents the variable's current value. Variables are given new values with *assignment operators*, *increment operators* and *decrement operators*. See Section 7.7 [Assignment Expressions], page 82.

A few variables have special built-in meanings, such as `FS`, the field separator, and `NF`, the number of fields in the current input record. See Chapter 10 [Built-in Variables], page 113, for a list of them. These built-in variables can be used and assigned just like all other variables, but their values

are also used or changed automatically by `awk`. All built-in variables names are entirely upper-case.

Variables in `awk` can be assigned either numeric or string values. By default, variables are initialized to the empty string, which is zero if converted to a number. There is no need to "initialize" each variable explicitly in `awk`, the way you would in C and in most other traditional languages.

7.3.2 Assigning Variables on the Command Line

You can set any `awk` variable by including a *variable assignment* among the arguments on the command line when you invoke `awk` (see Section 14.2 [Other Command Line Arguments], page 163). Such an assignment has this form:

variable=*text*

With it, you can set a variable either at the beginning of the `awk` run or in between input files.

If you precede the assignment with the '`-v`' option, like this:

`-v` *variable*=*text*

then the variable is set at the very beginning, before even the `BEGIN` rules are run. The '`-v`' option and its assignment must precede all the file name arguments, as well as the program text. (See Section 14.1 [Command Line Options], page 159, for more information about the '`-v`' option.)

Otherwise, the variable assignment is performed at a time determined by its position among the input file arguments: after the processing of the preceding input file argument. For example:

```
awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
```

prints the value of field number `n` for all input records. Before the first file is read, the command line sets the variable `n` equal to four. This causes the fourth field to be printed in lines from the file '`inventory-shipped`'. After the first file has finished, but before the second file is started, `n` is set to two, so that the second field is printed in lines from '`BBS-list`'.

```
$ awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
⊣ 15
⊣ 24
...
⊣ 555-5553
⊣ 555-3412
...
```

Command line arguments are made available for explicit examination by the `awk` program in an array named `ARGV` (see Section 10.3 [Using `ARGC` and `ARGV`], page 117).

`awk` processes the values of command line assignments for escape sequences (d.c.) (see Section 4.2 [Escape Sequences], page 24).

7.4 Conversion of Strings and Numbers

Strings are converted to numbers, and numbers to strings, if the context of the `awk` program demands it. For example, if the value of either `foo` or `bar` in the expression '`foo + bar`' happens to be a string, it is converted to a number before the addition is performed. If numeric values appear in string concatenation, they are converted to strings. Consider this:

```
two = 2; three = 3
print (two three) + 4
```

This prints the (numeric) value 27. The numeric values of the variables `two` and `three` are converted to strings and concatenated together, and the resulting string is converted back to the number 23, to which four is then added.

If, for some reason, you need to force a number to be converted to a string, concatenate the empty string, `""`, with that number. To force a string to be converted to a number, add zero to that string.

A string is converted to a number by interpreting any numeric prefix of the string as numerals: `"2.5"` converts to 2.5, `"1e3"` converts to 1000, and `"25fix"` has a numeric value of 25. Strings that can't be interpreted as valid numbers are converted to zero.

The exact manner in which numbers are converted into strings is controlled by the `awk` built-in variable `CONVFMT` (see Chapter 10 [Built-in Variables], page 113). Numbers are converted using the `sprintf` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135) with `CONVFMT` as the format specifier.

`CONVFMT`'s default value is `"%.6g"`, which prints a value with at least six significant digits. For some applications you will want to change it to specify more precision. On most modern machines, you must print 17 digits to capture a floating point number's value exactly.

Strange results can happen if you set `CONVFMT` to a string that doesn't tell `sprintf` how to format floating point numbers in a useful way. For example, if you forget the '`%`' in the format, all numbers will be converted to the same constant string.

As a special case, if a number is an integer, then the result of converting it to a string is *always* an integer, no matter what the value of `CONVFMT` may be. Given the following code fragment:

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

`b` has the value `"12"`, not `"12.00"` (d.c.).

Prior to the POSIX standard, `awk` specified that the value of `OFMT` was used for converting numbers to strings. `OFMT` specifies the output format to use when printing numbers with `print`. `CONVFMT` was introduced in order to separate the semantics of conversion from the semantics of printing. Both

`CONVFMT` and `OFMT` have the same default value: `"%.6g"`. In the vast majority of cases, old `awk` programs will not change their behavior. However, this use of `OFMT` is something to keep in mind if you must port your program to other implementations of `awk`; we recommend that instead of changing your programs, you just port `gawk` itself! See Section 6.1 [The `print` Statement], page 59, for more information on the `print` statement.

7.5 Arithmetic Operators

The `awk` language uses the common arithmetic operators when evaluating expressions. All of these arithmetic operators follow normal precedence rules, and work as you would expect them to. Arithmetic operations are evaluated using double precision floating point, which has the usual problems of inexactness and exceptions.[2]

Here is a file '`grades`' containing a list of student names and three test scores per student (it's a small class):

```
Pat   100 97 58
Sandy  84 72 93
Chris  72 92 89
```

This programs takes the file '`grades`', and prints the average of the scores.

```
$ awk '{ sum = $2 + $3 + $4 ; avg = sum / 3
>        print $1, avg }' grades
⊣ Pat 85
⊣ Sandy 83
⊣ Chris 84.3333
```

This table lists the arithmetic operators in `awk`, in order from highest precedence to lowest:

`- `*x* — Negation.

`+ `*x* — Unary plus. The expression is converted to a number.

x `^` *y*
x `**` *y* — Exponentiation: *x* raised to the *y* power. '`2 ^ 3`' has the value eight. The character sequence '`**`' is equivalent to '`^`'. (The POSIX standard only specifies the use of '`^`' for exponentiation.)

x `*` *y* — Multiplication.

x `/` *y* — Division. Since all numbers in `awk` are floating point numbers, the result is not rounded to an integer: '`3 / 4`' has the value 0.75.

---

[2] David Goldberg, *What Every Computer Scientist Should Know About Floating-point Arithmetic* (`http://www.validgh.com/goldberg/paper.ps`), *ACM Computing Surveys* **23**, 1 (1991-03), 5-48.

x `%` *y*
: Remainder. The quotient is rounded toward zero to an integer, multiplied by *y* and this result is subtracted from *x*. This operation is sometimes known as "trunc-mod." The following relation always holds:

```
b * int(a / b) + (a % b) == a
```

One possibly undesirable effect of this definition of remainder is that *x* `%` *y* is negative if *x* is negative. Thus,

```
-17 % 8 = -1
```

In other `awk` implementations, the signedness of the remainder may be machine dependent.

x `+` *y*
: Addition.

x `-` *y*
: Subtraction.

For maximum portability, do not use the '`**`' operator.

Unary plus and minus have the same precedence, the multiplication operators all have the same precedence, and addition and subtraction have the same precedence.

7.6 String Concatenation

It seemed like a good idea at the time.
Brian Kernighan

There is only one string operation: concatenation. It does not have a specific operator to represent it. Instead, concatenation is performed by writing expressions next to one another, with no operator. For example:

```
$ awk '{ print "Field number one: " $1 }' BBS-list
⊣ Field number one: aardvark
⊣ Field number one: alpo-net
...
```

Without the space in the string constant after the '`:`', the line would run together. For example:

```
$ awk '{ print "Field number one:" $1 }' BBS-list
⊣ Field number one:aardvark
⊣ Field number one:alpo-net
...
```

Since string concatenation does not have an explicit operator, it is often necessary to insure that it happens where you want it to by using parentheses to enclose the items to be concatenated. For example, the following code fragment does not concatenate `file` and `name` as you might expect:

```
file = "file"
name = "name"
print "something meaningful" > file name
```

It is necessary to use the following:

```
print "something meaningful" > (file name)
```

We recommend that you use parentheses around concatenation in all but the most common contexts (such as on the right-hand side of '`=`').

7.7 Assignment Expressions

An *assignment* is an expression that stores a new value into a variable. For example, let's assign the value one to the variable `z`:

```
z = 1
```

After this expression is executed, the variable `z` has the value one. Whatever old value `z` had before the assignment is forgotten.

Assignments can store string values also. For example, this would store the value `"this food is good"` in the variable `message`:

```
thing = "food"
predicate = "good"
message = "this " thing " is " predicate
```

(This also illustrates string concatenation.)

The '`=`' sign is called an *assignment operator*. It is the simplest assignment operator because the value of the right-hand operand is stored unchanged.

Most operators (addition, concatenation, and so on) have no effect except to compute a value. If you ignore the value, you might as well not use the operator. An assignment operator is different; it does produce a value, but even if you ignore the value, the assignment still makes itself felt through the alteration of the variable. We call this a *side effect*.

The left-hand operand of an assignment need not be a variable (see Section 7.3 [Variables], page 77); it can also be a field (see Section 5.4 [Changing the Contents of a Field], page 42) or an array element (see Chapter 11 [Arrays in `awk`], page 121). These are all called *lvalues*, which means they can appear on the left-hand side of an assignment operator. The right-hand operand may be any expression; it produces the new value which the assignment stores in the specified variable, field or array element. (Such values are called *rvalues*).

It is important to note that variables do *not* have permanent types. The type of a variable is simply the type of whatever value it happens to hold at the moment. In the following program fragment, the variable `foo` has a numeric value at first, and a string value later on:

```
foo = 1
print foo
foo = "bar"
print foo
```

When the second assignment gives `foo` a string value, the fact that it previously had a numeric value is forgotten.

String values that do not begin with a digit have a numeric value of zero. After executing this code, the value of `foo` is five:

```
foo = "a string"
foo = foo + 5
```

(Note that using a variable as a number and then later as a string can be confusing and is poor programming style. The above examples illustrate how `awk` works, *not* how you should write your own programs!)

An assignment is an expression, so it has a value: the same value that is assigned. Thus, '`z = 1`' as an expression has the value one. One consequence of this is that you can write multiple assignments together:

```
x = y = z = 0
```

stores the value zero in all three variables. It does this because the value of '`z = 0`', which is zero, is stored into `y`, and then the value of '`y = z = 0`', which is zero, is stored into `x`.

You can use an assignment anywhere an expression is called for. For example, it is valid to write '`x != (y = 1)`' to set `y` to one and then test whether `x` equals one. But this style tends to make programs hard to read; except in a one-shot program, you should not use such nesting of assignments.

Aside from '`=`', there are several other assignment operators that do arithmetic with the old value of the variable. For example, the operator '`+=`' computes a new value by adding the right-hand value to the old value of the variable. Thus, the following assignment adds five to the value of `foo`:

```
foo += 5
```

This is equivalent to the following:

```
foo = foo + 5
```

Use whichever one makes the meaning of your program clearer.

There are situations where using '`+=`' (or any assignment operator) is *not* the same as simply repeating the left-hand operand in the right-hand expression. For example:

```
# Thanks to Pat Rankin for this example
BEGIN  {
    foo[rand()] += 5
    for (x in foo)
       print x, foo[x]

    bar[rand()] = bar[rand()] + 5
    for (x in bar)
       print x, bar[x]
}
```

The indices of `bar` are guaranteed to be different, because `rand` will return different values each time it is called. (Arrays and the `rand` function haven't been covered yet. See Chapter 11 [Arrays in `awk`], page 121, and see Section 12.2 [Numeric Built-in Functions], page 134, for more information). This example illustrates an important fact about the assignment operators: the left-hand expression is only evaluated *once*.

It is also up to the implementation as to which expression is evaluated first, the left-hand one or the right-hand one. Consider this example:

```
i = 1
a[i += 2] = i + 1
```

The value of `a[3]` could be either two or four.

Here is a table of the arithmetic assignment operators. In each case, the right-hand operand is an expression whose value is converted to a number.

lvalue `+=` *increment*
: Adds *increment* to the value of *lvalue* to make the new value of *lvalue*.

lvalue `-=` *decrement*
: Subtracts *decrement* from the value of *lvalue*.

lvalue `*=` *coefficient*
: Multiplies the value of *lvalue* by *coefficient*.

lvalue `/=` *divisor*
: Divides the value of *lvalue* by *divisor*.

lvalue `%=` *modulus*
: Sets *lvalue* to its remainder by *modulus*.

lvalue `^=` *power*
lvalue `**=` *power*
: Raises *lvalue* to the power *power*. (Only the '`^=`' operator is specified by POSIX.)

For maximum portability, do not use the '`**=`' operator.

7.8 Increment and Decrement Operators

Increment and *decrement operators* increase or decrease the value of a variable by one. You could do the same thing with an assignment operator, so the increment operators add no power to the `awk` language; but they are convenient abbreviations for very common operations.

The operator to add one is written '`++`'. It can be used to increment a variable either before or after taking its value.

To pre-increment a variable *v*, write '`++`*v*'. This adds one to the value of *v* and that new value is also the value of this expression. The assignment expression '*v* `+= 1`' is completely equivalent.

Writing the '`++`' after the variable specifies post-increment. This increments the variable value just the same; the difference is that the value of the increment expression itself is the variable's *old* value. Thus, if `foo` has the value four, then the expression '`foo++`' has the value four, but it changes the value of `foo` to five.

The post-increment '`foo++`' is nearly equivalent to writing '`(foo += 1) - 1`'. It is not perfectly equivalent because all numbers in `awk` are floating point: in floating point, '`foo + 1 - 1`' does not necessarily equal `foo`. But the difference is minute as long as you stick to numbers that are fairly small (less than 10e12).

Any lvalue can be incremented. Fields and array elements are incremented just like variables. (Use '`$(i++)`' when you wish to do a field reference and a variable increment at the same time. The parentheses are necessary because of the precedence of the field reference operator, '`$`'.)

The decrement operator '`--`' works just like '`++`' except that it subtracts one instead of adding. Like '`++`', it can be used before the lvalue to predecrement or after it to post-decrement.

Here is a summary of increment and decrement expressions.

`++`*lvalue* This expression increments *lvalue* and the new value becomes the value of the expression.

lvalue`++` This expression increments *lvalue*, but the value of the expression is the *old* value of *lvalue*.

`--`*lvalue* Like '`++`*lvalue*', but instead of adding, it subtracts. It decrements *lvalue* and delivers the value that results.

lvalue`--` Like '*lvalue*`++`', but instead of adding, it subtracts. It decrements *lvalue*. The value of the expression is the *old* value of *lvalue*.

7.9 True and False in `awk`

Many programming languages have a special representation for the concepts of "true" and "false." Such languages usually use the special constants `true` and `false`, or perhaps their upper-case equivalents.

`awk` is different. It borrows a very simple concept of true and false from C. In `awk`, any non-zero numeric value, *or* any non-empty string value is true. Any other value (zero or the null string, `""`) is false. The following program will print '`A strange truth value`' three times:

```
BEGIN {
   if (3.1415927)
       print "A strange truth value"
   if ("Four Score And Seven Years Ago")
       print "A strange truth value"
   if (j = 57)
       print "A strange truth value"
}
```

There is a surprising consequence of the "non-zero or non-null" rule: The string constant `"0"` is actually true, since it is non-null (d.c.).

7.10 Variable Typing and Comparison Expressions

> *The Guide is definitive. Reality is frequently inaccurate.*
> The Hitchhiker's Guide to the Galaxy

Unlike other programming languages, `awk` variables do not have a fixed type. Instead, they can be either a number or a string, depending upon the value that is assigned to them.

The 1992 POSIX standard introduced the concept of a *numeric string*, which is simply a string that looks like a number, for example, `" +2"`. This concept is used for determining the type of a variable.

The type of the variable is important, since the types of two variables determine how they are compared.

In `gawk`, variable typing follows these rules.

1. A numeric literal or the result of a numeric operation has the *numeric* attribute.
2. A string literal or the result of a string operation has the *string* attribute.
3. Fields, `getline` input, `FILENAME`, `ARGV` elements, `ENVIRON` elements and the elements of an array created by `split` that are numeric strings have the *strnum* attribute. Otherwise, they have the *string* attribute. Uninitialized variables also have the *strnum* attribute.
4. Attributes propagate across assignments, but are not changed by any use.

The last rule is particularly important. In the following program, `a` has numeric type, even though it is later used in a string operation.

```
BEGIN {
        a = 12.345
        b = a " is a cute number"
        print b
}
```

When two operands are compared, either string comparison or numeric comparison may be used, depending on the attributes of the operands, according to the following, symmetric, matrix:

	STRING	NUMERIC	STRNUM
STRING	string	string	string
NUMERIC	string	numeric	numeric
STRNUM	string	numeric	numeric

The basic idea is that user input that looks numeric, and *only* user input, should be treated as numeric, even though it is actually made of characters, and is therefore also a string.

Comparison expressions compare strings or numbers for relationships such as equality. They are written using *relational operators*, which are a superset of those in C. Here is a table of them:

x `<` *y*
: True if *x* is less than *y*.

x `<=` *y*
: True if *x* is less than or equal to *y*.

x `>` *y*
: True if *x* is greater than *y*.

x `>=` *y*
: True if *x* is greater than or equal to *y*.

x `==` *y*
: True if *x* is equal to *y*.

x `!=` *y*
: True if *x* is not equal to *y*.

x `~` *y*
: True if the string *x* matches the regexp denoted by *y*.

x `!~` *y*
: True if the string *x* does not match the regexp denoted by *y*.

subscript `in` *array*
: True if the array *array* has an element with the subscript *subscript*.

Comparison expressions have the value one if true and zero if false.

When comparing operands of mixed types, numeric operands are converted to strings using the value of `CONVFMT` (see Section 7.4 [Conversion of Strings and Numbers], page 79).

Strings are compared by comparing the first character of each, then the second character of each, and so on. Thus `"10"` is less than `"9"`. If there are two strings where one is a prefix of the other, the shorter string is less than the longer one. Thus `"abc"` is less than `"abcd"`.

It is very easy to accidentally mistype the '`==`' operator, and leave off one of the '`=`'s. The result is still valid `awk` code, but the program will not do what you mean:

```
if (a = b)    # oops! should be a == b
   ...
else
   ...
```

Unless `b` happens to be zero or the null string, the `if` part of the test will always succeed. Because the operators are so similar, this kind of error is very difficult to spot when scanning the source code.

Here are some sample expressions, how `gawk` compares them, and what the result of the comparison is.

`1.5 <= 2.0`
: numeric comparison (true)

`"abc" >= "xyz"`
: string comparison (false)

`1.5 != " +2"`
: string comparison (true)

`"1e2" < "3"`
: string comparison (true)

```
a = 2; b = "2"
a == b
```
: string comparison (true)

```
a = 2; b = " +2"
a == b
```
: string comparison (false)

In this example,

```
$ echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
⊣ false
```

the result is '`false`' since both `$1` and `$2` are numeric strings and thus both have the *strnum* attribute, dictating a numeric comparison.

The purpose of the comparison rules and the use of numeric strings is to attempt to produce the behavior that is "least surprising," while still "doing the right thing."

String comparisons and regular expression comparisons are very different. For example,

```
x == "foo"
```

has the value of one, or is true, if the variable `x` is precisely '`foo`'. By contrast,

```
x ~ /foo/
```

has the value one if `x` contains '`foo`', such as `"Oh, what a fool am I!"`.

The right hand operand of the '`~`' and '`!~`' operators may be either a regexp constant (`/.../`), or an ordinary expression, in which case the value

of the expression as a string is used as a dynamic regexp (see Section 4.1 [How to Use Regular Expressions], page 23; also see Section 4.7 [Using Dynamic Regexps], page 35).

In recent implementations of `awk`, a constant regular expression in slashes by itself is also an expression. The regexp `/`*regexp*`/` is an abbreviation for this comparison expression:

```
$0 ~ /regexp/
```

One special place where `/foo/` is *not* an abbreviation for '`$0 ~ /foo/`' is when it is the right-hand operand of '`~`' or '`!~`'! See Section 7.2 [Using Regular Expression Constants], page 76, where this is discussed in more detail.

7.11 Boolean Expressions

A *boolean expression* is a combination of comparison expressions or matching expressions, using the boolean operators "or" ('`||`'), "and" ('`&&`'), and "not" ('`!`'), along with parentheses to control nesting. The truth value of the boolean expression is computed by combining the truth values of the component expressions. Boolean expressions are also referred to as *logical expressions*. The terms are equivalent.

Boolean expressions can be used wherever comparison and matching expressions can be used. They can be used in `if`, `while`, `do` and `for` statements (see Chapter 9 [Control Statements in Actions], page 103). They have numeric values (one if true, zero if false), which come into play if the result of the boolean expression is stored in a variable, or used in arithmetic.

In addition, every boolean expression is also a valid pattern, so you can use one as a pattern to control the execution of rules.

Here are descriptions of the three boolean operators, with examples.

boolean1 `&&` *boolean2*
: True if both *boolean1* and *boolean2* are true. For example, the following statement prints the current input record if it contains both '`2400`' and '`foo`'.

```
if ($0 ~ /2400/ && $0 ~ /foo/) print
```

The subexpression *boolean2* is evaluated only if *boolean1* is true. This can make a difference when *boolean2* contains expressions that have side effects: in the case of '`$0 ~ /foo/ && ($2 == bar++)`', the variable `bar` is not incremented if there is no '`foo`' in the record.

boolean1 `||` *boolean2*
: True if at least one of *boolean1* or *boolean2* is true. For example, the following statement prints all records in the input that contain *either* '`2400`' or '`foo`', or both.

```
if ($0 ~ /2400/ || $0 ~ /foo/) print
```

The subexpression *boolean2* is evaluated only if *boolean1* is false. This can make a difference when *boolean2* contains expressions that have side effects.

`!` *boolean* True if *boolean* is false. For example, the following program prints all records in the input file '`BBS-list`' that do *not* contain the string '`foo`'.

```
awk '{ if (! ($0 ~ /foo/)) print }' BBS-list
```

The '`&&`' and '`||`' operators are called *short-circuit* operators because of the way they work. Evaluation of the full expression is "short-circuited" if the result can be determined part way through its evaluation.

You can continue a statement that uses '`&&`' or '`||`' simply by putting a newline after them. But you cannot put a newline in front of either of these operators without using backslash continuation (see Section 2.6 [`awk` Statements Versus Lines], page 16).

The actual value of an expression using the '`!`' operator will be either one or zero, depending upon the truth value of the expression it is applied to.

The '`!`' operator is often useful for changing the sense of a flag variable from false to true and back again. For example, the following program is one way to print lines in between special bracketing lines:

```
$1 == "START"   { interested = ! interested }
interested == 1 { print }
$1 == "END"     { interested = ! interested }
```

The variable `interested`, like all `awk` variables, starts out initialized to zero, which is also false. When a line is seen whose first field is '`START`', the value of `interested` is toggled to true, using '`!`'. The next rule prints lines as long as `interested` is true. When a line is seen whose first field is '`END`', `interested` is toggled back to false.

7.12 Conditional Expressions

A *conditional expression* is a special kind of expression with three operands. It allows you to use one expression's value to select one of two other expressions.

The conditional expression is the same as in the C language:

selector `?` *if-true-exp* `:` *if-false-exp*

There are three subexpressions. The first, *selector*, is always computed first. If it is "true" (not zero and not null) then *if-true-exp* is computed next and its value becomes the value of the whole expression. Otherwise, *if-false-exp* is computed next and its value becomes the value of the whole expression.

For example, this expression produces the absolute value of `x`:

```
x > 0 ? x : -x
```

Each time the conditional expression is computed, exactly one of *if-true-exp* and *if-false-exp* is used; the other is ignored. This is important when

the expressions have side effects. For example, this conditional expression examines element `i` of either array `a` or array `b`, and increments `i`.

```
x == y ? a[i++] : b[i++]
```

This is guaranteed to increment `i` exactly once, because each time only one of the two increment expressions is executed, and the other is not. See Chapter 11 [Arrays in `awk`], page 121, for more information about arrays.

As a minor `gawk` extension, you can continue a statement that uses '`?:`' simply by putting a newline after either character. However, you cannot put a newline in front of either character without using backslash continuation (see Section 2.6 [`awk` Statements Versus Lines], page 16). If '`--posix`' is specified (see Section 14.1 [Command Line Options], page 159), then this extension is disabled.

7.13 Function Calls

A *function* is a name for a particular calculation. Because it has a name, you can ask for it by name at any point in the program. For example, the function `sqrt` computes the square root of a number.

A fixed set of functions are *built-in*, which means they are available in every `awk` program. The `sqrt` function is one of these. See Chapter 12 [Built-in Functions], page 133, for a list of built-in functions and their descriptions. In addition, you can define your own functions for use in your program. See Chapter 13 [User-defined Functions], page 151, for how to do this.

The way to use a function is with a *function call* expression, which consists of the function name followed immediately by a list of *arguments* in parentheses. The arguments are expressions which provide the raw materials for the function's calculations. When there is more than one argument, they are separated by commas. If there are no arguments, write just '`()`' after the function name. Here are some examples:

`sqrt(x^2 + y^2)`	*one argument*
`atan2(y, x)`	*two arguments*
`rand()`	*no arguments*

Do not put any space between the function name and the open-parenthesis! A user-defined function name looks just like the name of a variable, and space would make the expression look like concatenation of a variable with an expression inside parentheses. Space before the parenthesis is harmless with built-in functions, but it is best not to get into the habit of using space to avoid mistakes with user-defined functions.

Each function expects a particular number of arguments. For example, the `sqrt` function must be called with a single argument, the number to take the square root of:

`sqrt(`*argument*`)`

Some of the built-in functions allow you to omit the final argument. If you do so, they use a reasonable default. See Chapter 12 [Built-in Functions],

page 133, for full details. If arguments are omitted in calls to user-defined functions, then those arguments are treated as local variables, initialized to the empty string (see Chapter 13 [User-defined Functions], page 151).

Like every other expression, the function call has a value, which is computed by the function based on the arguments you give it. In this example, the value of '`sqrt(`*argument*`)`' is the square root of *argument*. A function can also have side effects, such as assigning values to certain variables or doing I/O.

Here is a command to read numbers, one number per line, and print the square root of each one:

```
$ awk '{ print "The square root of", $1, "is", sqrt($1) }'
1
⊣ The square root of 1 is 1
3
⊣ The square root of 3 is 1.73205
5
⊣ The square root of 5 is 2.23607
Control-d
```

7.14 Operator Precedence (How Operators Nest)

Operator precedence determines how operators are grouped, when different operators appear close by in one expression. For example, '`*`' has higher precedence than '`+`'; thus, '`a + b * c`' means to multiply `b` and `c`, and then add `a` to the product (i.e. '`a + (b * c)`').

You can overrule the precedence of the operators by using parentheses. You can think of the precedence rules as saying where the parentheses are assumed to be if you do not write parentheses yourself. In fact, it is wise to always use parentheses whenever you have an unusual combination of operators, because other people who read the program may not remember what the precedence is in this case. You might forget, too; then you could make a mistake. Explicit parentheses will help prevent any such mistake.

When operators of equal precedence are used together, the leftmost operator groups first, except for the assignment, conditional and exponentiation operators, which group in the opposite order. Thus, '`a - b + c`' groups as '`(a - b) + c`', and '`a = b = c`' groups as '`a = (b = c)`'.

The precedence of prefix unary operators does not matter as long as only unary operators are involved, because there is only one way to interpret them—innermost first. Thus, '`$++i`' means '`$(++i)`' and '`++$x`' means '`++($x)`'. However, when another operator follows the operand, then the precedence of the unary operators can matter. Thus, '`$x^2`' means '`($x)^2`', but '`-x^2`' means '`-(x^2)`', because '`-`' has lower precedence than '`^`' while '`$`' has higher precedence.

Here is a table of `awk`'s operators, in order from highest precedence to lowest:

`(...)` Grouping.

`$` Field.

`++ --` Increment, decrement.

`^ **` Exponentiation. These operators group right-to-left. (The '`**`' operator is not specified by POSIX.)

`+ - !` Unary plus, minus, logical "not".

`* / %` Multiplication, division, modulus.

`+ -` Addition, subtraction.

Concatenation
No special token is used to indicate concatenation. The operands are simply written side by side.

`< <= == !=`
`> >= >> |` Relational, and redirection. The relational operators and the redirections have the same precedence level. Characters such as '`>`' serve both as relationals and as redirections; the context distinguishes between the two meanings.

Note that the I/O redirection operators in `print` and `printf` statements belong to the statement level, not to expressions. The redirection does not produce an expression which could be the operand of another operator. As a result, it does not make sense to use a redirection operator near another operator of lower precedence, without parentheses. Such combinations, for example '`print foo > a ? b : c`', result in syntax errors. The correct way to write this statement is '`print foo > (a ? b : c)`'.

`~ !~` Matching, non-matching.

`in` Array membership.

`&&` Logical "and".

`||` Logical "or".

`?:` Conditional. This operator groups right-to-left.

`= += -= *=`
`/= %= ^= **=`
Assignment. These operators group right-to-left. (The '`**=`' operator is not specified by POSIX.)

8 Patterns and Actions

As you have already seen, each `awk` statement consists of a pattern with an associated action. This chapter describes how you build patterns and actions.

8.1 Pattern Elements

Patterns in `awk` control the execution of rules: a rule is executed when its pattern matches the current input record. This section explains all about how to write patterns.

8.1.1 Kinds of Patterns

Here is a summary of the types of patterns supported in `awk`.

/regular expression/
: A regular expression as a pattern. It matches when the text of the input record fits the regular expression. (See Chapter 4 [Regular Expressions], page 23.)

expression
: A single expression. It matches when its value is non-zero (if a number) or non-null (if a string). (See Section 8.1.3 [Expressions as Patterns], page 96.)

pat1`,` *pat2*
: A pair of patterns separated by a comma, specifying a range of records. The range includes both the initial record that matches *pat1*, and the final record that matches *pat2*. (See Section 8.1.4 [Specifying Record Ranges with Patterns], page 97.)

`BEGIN`
`END`
: Special patterns for you to supply start-up or clean-up actions for your `awk` program. (See Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98.)

empty
: The empty pattern matches every input record. (See Section 8.1.6 [The Empty Pattern], page 100.)

8.1.2 Regular Expressions as Patterns

We have been using regular expressions as patterns since our early examples. This kind of pattern is simply a regexp constant in the pattern part of a rule. Its meaning is '`$0 ~ /`*pattern*`/`'. The pattern matches when the input record matches the regexp. For example:

```
/foo|bar|baz/  { buzzwords++ }
END            { print buzzwords, "buzzwords seen" }
```

8.1.3 Expressions as Patterns

Any `awk` expression is valid as an `awk` pattern. Then the pattern matches if the expression's value is non-zero (if a number) or non-null (if a string).

The expression is reevaluated each time the rule is tested against a new input record. If the expression uses fields such as `$1`, the value depends directly on the new input record's text; otherwise, it depends only on what has happened so far in the execution of the `awk` program, but that may still be useful.

A very common kind of expression used as a pattern is the comparison expression, using the comparison operators described in Section 7.10 [Variable Typing and Comparison Expressions], page 86.

Regexp matching and non-matching are also very common expressions. The left operand of the '`~`' and '`!~`' operators is a string. The right operand is either a constant regular expression enclosed in slashes (*/regexp/*), or any expression, whose string value is used as a dynamic regular expression (see Section 4.7 [Using Dynamic Regexps], page 35).

The following example prints the second field of each input record whose first field is precisely '`foo`'.

```
$ awk '$1 == "foo" { print $2 }' BBS-list
```

(There is no output, since there is no BBS site named "foo".) Contrast this with the following regular expression match, which would accept any record with a first field that contains '`foo`':

```
$ awk '$1 ~ /foo/ { print $2 }' BBS-list
⊣ 555-1234
⊣ 555-6699
⊣ 555-6480
⊣ 555-2127
```

Boolean expressions are also commonly used as patterns. Whether the pattern matches an input record depends on whether its subexpressions match.

For example, the following command prints all records in '`BBS-list`' that contain both '`2400`' and '`foo`'.

```
$ awk '/2400/ && /foo/' BBS-list
⊣ fooey        555-1234     2400/1200/300     B
```

The following command prints all records in '`BBS-list`' that contain *either* '`2400`' or '`foo`', or both.

```
$ awk '/2400/ || /foo/' BBS-list
 ⊣ alpo-net     555-3412     2400/1200/300     A
 ⊣ bites        555-1675     2400/1200/300     A
 ⊣ fooey        555-1234     2400/1200/300     B
 ⊣ foot         555-6699     1200/300          B
 ⊣ macfoo       555-6480     1200/300          A
 ⊣ sdace        555-3430     2400/1200/300     A
 ⊣ sabafoo      555-2127     1200/300          C
```

The following command prints all records in '`BBS-list`' that do *not* contain the string '`foo`'.

```
$ awk '! /foo/' BBS-list
 ⊣ aardvark     555-5553     1200/300          B
 ⊣ alpo-net     555-3412     2400/1200/300     A
 ⊣ barfly       555-7685     1200/300          A
 ⊣ bites        555-1675     2400/1200/300     A
 ⊣ camelot      555-0542     300               C
 ⊣ core         555-2912     1200/300          C
 ⊣ sdace        555-3430     2400/1200/300     A
```

The subexpressions of a boolean operator in a pattern can be constant regular expressions, comparisons, or any other `awk` expressions. Range patterns are not expressions, so they cannot appear inside boolean patterns. Likewise, the special patterns `BEGIN` and `END`, which never match any input record, are not expressions and cannot appear inside boolean patterns.

A regexp constant as a pattern is also a special case of an expression pattern. `/foo/` as an expression has the value one if '`foo`' appears in the current input record; thus, as a pattern, `/foo/` matches any record containing '`foo`'.

8.1.4 Specifying Record Ranges with Patterns

A *range pattern* is made of two patterns separated by a comma, of the form '*begpat*`,` *endpat*'. It matches ranges of consecutive input records. The first pattern, *begpat*, controls where the range begins, and the second one, *endpat*, controls where it ends. For example,

```
awk '$1 == "on", $1 == "off"'
```

prints every record between '`on`'/'`off`' pairs, inclusive.

A range pattern starts out by matching *begpat* against every input record; when a record matches *begpat*, the range pattern becomes *turned on*. The range pattern matches this record. As long as it stays turned on, it automatically matches every input record read. It also matches *endpat* against every input record; when that succeeds, the range pattern is turned off again for the following record. Then it goes back to checking *begpat* against each record.

The record that turns on the range pattern and the one that turns it off both match the range pattern. If you don't want to operate on these records,

you can write `if` statements in the rule's action to distinguish them from the records you are interested in.

It is possible for a pattern to be turned both on and off by the same record, if the record satisfies both conditions. Then the action is executed for just that record.

For example, suppose you have text between two identical markers (say the '`%`' symbol) that you wish to ignore. You might try to combine a range pattern that describes the delimited text with the `next` statement (not discussed yet, see Section 9.7 [The `next` Statement], page 108), which causes `awk` to skip any further processing of the current record and start over again with the next input record. Such a program would look like this:

```
/^%$/,/^%$/    { next }
               { print }
```

This program fails because the range pattern is both turned on and turned off by the first line with just a '`%`' on it. To accomplish this task, you must write the program this way, using a flag:

```
/^%$/     { skip = ! skip; next }
skip == 1 { next } # skip lines with 'skip' set
```

Note that in a range pattern, the '`,`' has the lowest precedence (is evaluated last) of all the operators. Thus, for example, the following program attempts to combine a range pattern with another, simpler test.

```
echo Yes | awk '/1/,/2/ || /Yes/'
```

The author of this program intended it to mean '`(/1/,/2/) || /Yes/`'. However, `awk` interprets this as '`/1/, (/2/ || /Yes/)`'. This cannot be changed or worked around; range patterns do not combine with other patterns.

8.1.5 The `BEGIN` and `END` Special Patterns

`BEGIN` and `END` are special patterns. They are not used to match input records. Rather, they supply start-up or clean-up actions for your `awk` script.

8.1.5.1 Startup and Cleanup Actions

A `BEGIN` rule is executed, once, before the first input record has been read. An `END` rule is executed, once, after all the input has been read. For example:

```
$ awk '
> BEGIN { print "Analysis of \"foo\"" }
> /foo/ { ++n }
> END   { print "\"foo\" appears " n " times." }' BBS-list
⊣ Analysis of "foo"
⊣ "foo" appears 4 times.
```

This program finds the number of records in the input file 'BBS-list' that contain the string 'foo'. The BEGIN rule prints a title for the report. There is no need to use the BEGIN rule to initialize the counter n to zero, as awk does this automatically (see Section 7.3 [Variables], page 77).

The second rule increments the variable n every time a record containing the pattern 'foo' is read. The END rule prints the value of n at the end of the run.

The special patterns BEGIN and END cannot be used in ranges or with boolean operators (indeed, they cannot be used with any operators).

An awk program may have multiple BEGIN and/or END rules. They are executed in the order they appear, all the BEGIN rules at start-up and all the END rules at termination. BEGIN and END rules may be intermixed with other rules. This feature was added in the 1987 version of awk, and is included in the POSIX standard. The original (1978) version of awk required you to put the BEGIN rule at the beginning of the program, and the END rule at the end, and only allowed one of each. This is no longer required, but it is a good idea in terms of program organization and readability.

Multiple BEGIN and END rules are useful for writing library functions, since each library file can have its own BEGIN and/or END rule to do its own initialization and/or cleanup. Note that the order in which library functions are named on the command line controls the order in which their BEGIN and END rules are executed. Therefore you have to be careful to write such rules in library files so that the order in which they are executed doesn't matter. See Section 14.1 [Command Line Options], page 159, for more information on using library functions. See Chapter 15 [A Library of awk Functions], page 167, for a number of useful library functions.

If an awk program only has a BEGIN rule, and no other rules, then the program exits after the BEGIN rule has been run. (The original version of awk used to keep reading and ignoring input until end of file was seen.) However, if an END rule exists, then the input will be read, even if there are no other rules in the program. This is necessary in case the END rule checks the FNR and NR variables (d.c.).

BEGIN and END rules must have actions; there is no default action for these rules since there is no current record when they run.

8.1.5.2 Input/Output from BEGIN and END Rules

There are several (sometimes subtle) issues involved when doing I/O from a BEGIN or END rule.

The first has to do with the value of $0 in a BEGIN rule. Since BEGIN rules are executed before any input is read, there simply is no input record, and therefore no fields, when executing BEGIN rules. References to $0 and the fields yield a null string or zero, depending upon the context. One way to give $0 a real value is to execute a getline command without a variable

(see Section 5.8 [Explicit Input with `getline`], page 53). Another way is to simply assign a value to it.

The second point is similar to the first, but from the other direction. Inside an `END` rule, what is the value of `$0` and `NF`? Traditionally, due largely to implementation issues, `$0` and `NF` were *undefined* inside an `END` rule. The POSIX standard specified that `NF` was available in an `END` rule, containing the number of fields from the last input record. Due most probably to an oversight, the standard does not say that `$0` is also preserved, although logically one would think that it should be. In fact, `gawk` does preserve the value of `$0` for use in `END` rules. Be aware, however, that Unix `awk`, and possibly other implementations, do not.

The third point follows from the first two. What is the meaning of '`print`' inside a `BEGIN` or `END` rule? The meaning is the same as always, '`print $0`'. If `$0` is the null string, then this prints an empty line. Many long time `awk` programmers use '`print`' in `BEGIN` and `END` rules, to mean '`print ""`', relying on `$0` being null. While you might generally get away with this in `BEGIN` rules, in `gawk` at least, it is a very bad idea in `END` rules. It is also poor style, since if you want an empty line in the output, you should say so explicitly in your program.

8.1.6 The Empty Pattern

An empty (i.e. non-existent) pattern is considered to match *every* input record. For example, the program:

```
awk '{ print $1 }' BBS-list
```

prints the first field of every record.

8.2 Overview of Actions

An `awk` program or script consists of a series of rules and function definitions, interspersed. (Functions are described later. See Chapter 13 [User-defined Functions], page 151.)

A rule contains a pattern and an action, either of which (but not both) may be omitted. The purpose of the *action* is to tell `awk` what to do once a match for the pattern is found. Thus, in outline, an `awk` program generally looks like this:

```
[pattern] [{ action }]
[pattern] [{ action }]
...
function name(args) { ... }
...
```

An action consists of one or more `awk` *statements*, enclosed in curly braces ('`{`' and '`}`'). Each statement specifies one thing to be done. The statements are separated by newlines or semicolons.

The curly braces around an action must be used even if the action contains only one statement, or even if it contains no statements at all. However, if you omit the action entirely, omit the curly braces as well. An omitted action is equivalent to '`{ print $0 }`'.

```
/foo/  { }  # match foo, do nothing - empty action
/foo/       # match foo, print the record - omitted action
```

Here are the kinds of statements supported in `awk`:

- Expressions, which can call functions or assign values to variables (see Chapter 7 [Expressions], page 75). Executing this kind of statement simply computes the value of the expression. This is useful when the expression has side effects (see Section 7.7 [Assignment Expressions], page 82).
- Control statements, which specify the control flow of `awk` programs. The `awk` language gives you C-like constructs (`if`, `for`, `while`, and `do`) as well as a few special ones (see Chapter 9 [Control Statements in Actions], page 103).
- Compound statements, which consist of one or more statements enclosed in curly braces. A compound statement is used in order to put several statements together in the body of an `if`, `while`, `do` or `for` statement.
- Input statements, using the `getline` command (see Section 5.8 [Explicit Input with `getline`], page 53), the `next` statement (see Section 9.7 [The `next` Statement], page 108), and the `nextfile` statement (see Section 9.8 [The `nextfile` Statement], page 109).
- Output statements, `print` and `printf`. See Chapter 6 [Printing Output], page 59.
- Deletion statements, for deleting array elements. See Section 11.6 [The `delete` Statement], page 125.

The next chapter covers control statements in detail.

9 Control Statements in Actions

Control statements such as `if`, `while`, and so on control the flow of execution in `awk` programs. Most of the control statements in `awk` are patterned on similar statements in C.

All the control statements start with special keywords such as `if` and `while`, to distinguish them from simple expressions.

Many control statements contain other statements; for example, the `if` statement contains another statement which may or may not be executed. The contained statement is called the *body*. If you want to include more than one statement in the body, group them into a single *compound statement* with curly braces, separating them with newlines or semicolons.

9.1 The `if-else` Statement

The `if-else` statement is `awk`'s decision-making statement. It looks like this:

`if (`*condition*`)` *then-body* [`else` *else-body*]

The *condition* is an expression that controls what the rest of the statement will do. If *condition* is true, *then-body* is executed; otherwise, *else-body* is executed. The `else` part of the statement is optional. The condition is considered false if its value is zero or the null string, and true otherwise.

Here is an example:

```
if (x % 2 == 0)
    print "x is even"
else
    print "x is odd"
```

In this example, if the expression '`x % 2 == 0`' is true (that is, the value of `x` is evenly divisible by two), then the first `print` statement is executed, otherwise the second `print` statement is executed.

If the `else` appears on the same line as *then-body*, and *then-body* is not a compound statement (i.e. not surrounded by curly braces), then a semicolon must separate *then-body* from `else`. To illustrate this, let's rewrite the previous example:

```
if (x % 2 == 0) print "x is even"; else
        print "x is odd"
```

If you forget the '`;`', `awk` won't be able to interpret the statement, and you will get a syntax error.

We would not actually write this example this way, because a human reader might fail to see the `else` if it were not the first thing on its line.

9.2 The `while` Statement

In programming, a *loop* means a part of a program that can be executed two or more times in succession.

The `while` statement is the simplest looping statement in `awk`. It repeatedly executes a statement as long as a condition is true. It looks like this:

```
while (condition)
  body
```

Here *body* is a statement that we call the *body* of the loop, and *condition* is an expression that controls how long the loop keeps running.

The first thing the `while` statement does is test *condition*. If *condition* is true, it executes the statement *body*. After *body* has been executed, *condition* is tested again, and if it is still true, *body* is executed again. This process repeats until *condition* is no longer true. If *condition* is initially false, the body of the loop is never executed, and `awk` continues with the statement following the loop.

This example prints the first three fields of each record, one per line.

```
awk '{ i = 1
       while (i <= 3) {
           print $i
           i++
       }
}' inventory-shipped
```

Here the body of the loop is a compound statement enclosed in braces, containing two statements.

The loop works like this: first, the value of `i` is set to one. Then, the `while` tests whether `i` is less than or equal to three. This is true when `i` equals one, so the `i`-th field is printed. Then the '`i++`' increments the value of `i` and the loop repeats. The loop terminates when `i` reaches four.

As you can see, a newline is not required between the condition and the body; but using one makes the program clearer unless the body is a compound statement or is very simple. The newline after the open-brace that begins the compound statement is not required either, but the program would be harder to read without it.

9.3 The `do-while` Statement

The `do` loop is a variation of the `while` looping statement. The `do` loop executes the *body* once, and then repeats *body* as long as *condition* is true. It looks like this:

```
do
  body
while (condition)
```

Even if *condition* is false at the start, *body* is executed at least once (and only once, unless executing *body* makes *condition* true). Contrast this with the corresponding `while` statement:

```
while (condition)
  body
```

This statement does not execute *body* even once if *condition* is false to begin with.

Here is an example of a `do` statement:

```
awk '{ i = 1
       do {
          print $0
          i++
       } while (i <= 10)
}'
```

This program prints each input record ten times. It isn't a very realistic example, since in this case an ordinary `while` would do just as well. But this reflects actual experience; there is only occasionally a real use for a `do` statement.

9.4 The `for` Statement

The `for` statement makes it more convenient to count iterations of a loop. The general form of the `for` statement looks like this:

```
for (initialization; condition; increment)
  body
```

The *initialization*, *condition* and *increment* parts are arbitrary `awk` expressions, and *body* stands for any `awk` statement.

The `for` statement starts by executing *initialization*. Then, as long as *condition* is true, it repeatedly executes *body* and then *increment*. Typically *initialization* sets a variable to either zero or one, *increment* adds one to it, and *condition* compares it against the desired number of iterations.

Here is an example of a `for` statement:

```
awk '{ for (i = 1; i <= 3; i++)
          print $i
}' inventory-shipped
```

This prints the first three fields of each input record, one field per line.

You cannot set more than one variable in the *initialization* part unless you use a multiple assignment statement such as '`x = y = 0`', which is possible only if all the initial values are equal. (But you can initialize additional variables by writing their assignments as separate statements preceding the `for` loop.)

The same is true of the *increment* part; to increment additional variables, you must write separate statements at the end of the loop. The C compound

expression, using C's comma operator, would be useful in this context, but it is not supported in `awk`.

Most often, *increment* is an increment expression, as in the example above. But this is not required; it can be any expression whatever. For example, this statement prints all the powers of two between one and 100:

```
for (i = 1; i <= 100; i *= 2)
  print i
```

Any of the three expressions in the parentheses following the `for` may be omitted if there is nothing to be done there. Thus, '`for (; x > 0;)`' is equivalent to '`while (x > 0)`'. If the *condition* is omitted, it is treated as *true*, effectively yielding an *infinite loop* (i.e. a loop that will never terminate).

In most cases, a `for` loop is an abbreviation for a `while` loop, as shown here:

```
initialization
while (condition) {
  body
  increment
}
```

The only exception is when the `continue` statement (see Section 9.6 [The `continue` Statement], page 107) is used inside the loop; changing a `for` statement to a `while` statement in this way can change the effect of the `continue` statement inside the loop.

There is an alternate version of the `for` loop, for iterating over all the indices of an array:

```
for (i in array)
    do something with array[i]
```

See Section 11.5 [Scanning All Elements of an Array], page 124, for more information on this version of the `for` loop.

The `awk` language has a `for` statement in addition to a `while` statement because often a `for` loop is both less work to type and more natural to think of. Counting the number of iterations is very common in loops. It can be easier to think of this counting as part of looping rather than as something to do inside the loop.

The next section has more complicated examples of `for` loops.

9.5 The `break` Statement

The `break` statement jumps out of the innermost `for`, `while`, or `do` loop that encloses it. The following example finds the smallest divisor of any integer, and also identifies prime numbers:

```
awk '# find smallest divisor of num
     { num = $1
```

```
        for (div = 2; div*div <= num; div++)
          if (num % div == 0)
            break
        if (num % div == 0)
          printf "Smallest divisor of %d is %d\n", num, div
        else
          printf "%d is prime\n", num
      }'
```

When the remainder is zero in the first `if` statement, `awk` immediately *breaks out* of the containing `for` loop. This means that `awk` proceeds immediately to the statement following the loop and continues processing. (This is very different from the `exit` statement which stops the entire `awk` program. See Section 9.9 [The `exit` Statement], page 110.)

Here is another program equivalent to the previous one. It illustrates how the *condition* of a `for` or `while` could just as well be replaced with a `break` inside an `if`:

```
awk '# find smallest divisor of num
     { num = $1
       for (div = 2; ; div++) {
         if (num % div == 0) {
           printf "Smallest divisor of %d is %d\n", num, div
           break
         }
         if (div*div > num) {
           printf "%d is prime\n", num
           break
         }
       }
}'
```

As described above, the `break` statement has no meaning when used outside the body of a loop. However, although it was never documented, historical implementations of `awk` have treated the `break` statement outside of a loop as if it were a `next` statement (see Section 9.7 [The `next` Statement], page 108). Recent versions of Unix `awk` no longer allow this usage. `gawk` will support this use of `break` only if '`--traditional`' has been specified on the command line (see Section 14.1 [Command Line Options], page 159). Otherwise, it will be treated as an error, since the POSIX standard specifies that `break` should only be used inside the body of a loop (d.c.).

9.6 The `continue` Statement

The `continue` statement, like `break`, is used only inside `for`, `while`, and `do` loops. It skips over the rest of the loop body, causing the next cycle

around the loop to begin immediately. Contrast this with `break`, which jumps out of the loop altogether.

The `continue` statement in a `for` loop directs `awk` to skip the rest of the body of the loop, and resume execution with the increment-expression of the `for` statement. The following program illustrates this fact:

```
awk 'BEGIN {
     for (x = 0; x <= 20; x++) {
         if (x == 5)
             continue
         printf "%d ", x
     }
     print ""
}'
```

This program prints all the numbers from zero to 20, except for five, for which the `printf` is skipped. Since the increment '`x++`' is not skipped, `x` does not remain stuck at five. Contrast the `for` loop above with this `while` loop:

```
awk 'BEGIN {
     x = 0
     while (x <= 20) {
         if (x == 5)
             continue
         printf "%d ", x
         x++
     }
     print ""
}'
```

This program loops forever once `x` gets to five.

As described above, the `continue` statement has no meaning when used outside the body of a loop. However, although it was never documented, historical implementations of `awk` have treated the `continue` statement outside of a loop as if it were a `next` statement (see Section 9.7 [The `next` Statement], page 108). Recent versions of Unix `awk` no longer allow this usage. `gawk` will support this use of `continue` only if '`--traditional`' has been specified on the command line (see Section 14.1 [Command Line Options], page 159). Otherwise, it will be treated as an error, since the POSIX standard specifies that `continue` should only be used inside the body of a loop (d.c.).

9.7 The `next` Statement

The `next` statement forces `awk` to immediately stop processing the current record and go on to the next record. This means that no further rules are

executed for the current record. The rest of the current rule's action is not executed either.

Contrast this with the effect of the `getline` function (see Section 5.8 [Explicit Input with `getline`], page 53). That too causes `awk` to read the next record immediately, but it does not alter the flow of control in any way. So the rest of the current action executes with a new input record.

At the highest level, `awk` program execution is a loop that reads an input record and then tests each rule's pattern against it. If you think of this loop as a `for` statement whose body contains the rules, then the `next` statement is analogous to a `continue` statement: it skips to the end of the body of this implicit loop, and executes the increment (which reads another record).

For example, if your `awk` program works only on records with four fields, and you don't want it to fail when given bad input, you might use this rule near the beginning of the program:

```
NF != 4 {
  err = sprintf("%s:%d: skipped: NF != 4\n", FILENAME, FNR)
  print err > "/dev/stderr"
  next
}
```

so that the following rules will not see the bad record. The error message is redirected to the standard error output stream, as error messages should be. See Section 6.7 [Special File Names in `gawk`], page 69.

According to the POSIX standard, the behavior is undefined if the `next` statement is used in a `BEGIN` or `END` rule. `gawk` will treat it as a syntax error. Although POSIX permits it, some other `awk` implementations don't allow the `next` statement inside function bodies (see Chapter 13 [User-defined Functions], page 151). Just as any other `next` statement, a `next` inside a function body reads the next record and starts processing it with the first rule in the program.

If the `next` statement causes the end of the input to be reached, then the code in any `END` rules will be executed. See Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98.

Caution: Some `awk` implementations generate a run-time error if you use the `next` statement inside a user-defined function (see Chapter 13 [User-defined Functions], page 151). `gawk` does not have this problem.

9.8 The `nextfile` Statement

`gawk` provides the `nextfile` statement, which is similar to the `next` statement. However, instead of abandoning processing of the current record, the `nextfile` statement instructs `gawk` to stop processing the current data file.

Upon execution of the `nextfile` statement, `FILENAME` is updated to the name of the next data file listed on the command line, `FNR` is reset to one,

`ARGIND` is incremented, and processing starts over with the first rule in the progam. See Chapter 10 [Built-in Variables], page 113.

If the `nextfile` statement causes the end of the input to be reached, then the code in any `END` rules will be executed. See Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98.

The `nextfile` statement is a `gawk` extension; it is not (currently) available in any other `awk` implementation. See Section 15.2 [Implementing `nextfile` as a Function], page 167, for a user-defined function you can use to simulate the `nextfile` statement.

The `nextfile` statement would be useful if you have many data files to process, and you expect that you would not want to process every record in every file. Normally, in order to move on to the next data file, you would have to continue scanning the unwanted records. The `nextfile` statement accomplishes this much more efficiently.

Caution: Versions of `gawk` prior to 3.0 used two words ('`next file`') for the `nextfile` statement. This was changed in 3.0 to one word, since the treatment of '`file`' was inconsistent. When it appeared after `next`, it was a keyword. Otherwise, it was a regular identifier. The old usage is still accepted. However, `gawk` will generate a warning message, and support for `next file` will eventually be discontinued in a future version of `gawk`.

9.9 The `exit` Statement

The `exit` statement causes `awk` to immediately stop executing the current rule and to stop processing input; any remaining input is ignored. It looks like this:

`exit` [*return code*]

If an `exit` statement is executed from a `BEGIN` rule the program stops processing everything immediately. No input records are read. However, if an `END` rule is present, it is executed (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98).

If `exit` is used as part of an `END` rule, it causes the program to stop immediately.

An `exit` statement that is not part of a `BEGIN` or `END` rule stops the execution of any further automatic rules for the current record, skips reading any remaining input records, and executes the `END` rule if there is one.

If you do not want the `END` rule to do its job in this case, you can set a variable to non-zero before the `exit` statement, and check that variable in the `END` rule. See Section 15.3 [Assertions], page 169, for an example that does this.

If an argument is supplied to `exit`, its value is used as the exit status code for the `awk` process. If no argument is supplied, `exit` returns status zero (success). In the case where an argument is supplied to a first `exit`

statement, and then `exit` is called a second time with no argument, the previously supplied exit value is used (d.c.).

For example, let's say you've discovered an error condition you really don't know how to handle. Conventionally, programs report this by exiting with a non-zero status. Your `awk` program can do this using an `exit` statement with a non-zero argument. Here is an example:

```
BEGIN {
       if (("date" | getline date_now) <= 0) {
         print "Can't get system date" > "/dev/stderr"
         exit 1
       }
       print "current date is", date_now
       close("date")
}
```

10 Built-in Variables

Most `awk` variables are available for you to use for your own purposes; they never change except when your program assigns values to them, and never affect anything except when your program examines them. However, a few variables in `awk` have special built-in meanings. Some of them `awk` examines automatically, so that they enable you to tell `awk` how to do certain things. Others are set automatically by `awk`, so that they carry information from the internal workings of `awk` to your program.

This chapter documents all the built-in variables of `gawk`. Most of them are also documented in the chapters describing their areas of activity.

10.1 Built-in Variables that Control `awk`

This is an alphabetical list of the variables which you can change to control how `awk` does certain things. Those variables that are specific to `gawk` are marked with an asterisk, '`*`'.

`CONVFMT` This string controls conversion of numbers to strings (see Section 7.4 [Conversion of Strings and Numbers], page 79). It works by being passed, in effect, as the first argument to the `sprintf` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135). Its default value is `"%.6g"`. `CONVFMT` was introduced by the POSIX standard.

`FIELDWIDTHS` `*`
: This is a space separated list of columns that tells `gawk` how to split input with fixed, columnar boundaries. It is an experimental feature. Assigning to `FIELDWIDTHS` overrides the use of `FS` for field splitting. See Section 5.6 [Reading Fixed-width Data], page 49, for more information.

  If `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159), then `FIELDWIDTHS` has no special meaning, and field splitting operations are done based exclusively on the value of `FS`.

`FS` `FS` is the input field separator (see Section 5.5 [Specifying How Fields are Separated], page 44). The value is a single-character string or a multi-character regular expression that matches the separations between fields in an input record. If the value is the null string (`""`), then each character in the record becomes a separate field.

  The default value is `" "`, a string consisting of a single space. As a special exception, this value means that any sequence of spaces, tabs, and/or newlines is a single separator.[1] It also

[1] In POSIX `awk`, newline does not count as whitespace.

causes spaces, tabs, and newlines at the beginning and end of a record to be ignored.

You can set the value of `FS` on the command line using the '`-F`' option:

```
awk -F, 'program' input-files
```

If `gawk` is using `FIELDWIDTHS` for field-splitting, assigning a value to `FS` will cause `gawk` to return to the normal, `FS`-based, field splitting. An easy way to do this is to simply say '`FS = FS`', perhaps with an explanatory comment.

`IGNORECASE *`

If `IGNORECASE` is non-zero or non-null, then all string comparisons, and all regular expression matching are case-independent. Thus, regexp matching with '`~`' and '`!~`', and the `gensub`, `gsub`, `index`, `match`, `split` and `sub` functions, record termination with `RS`, and field splitting with `FS` all ignore case when doing their particular regexp operations. The value of `IGNORECASE` does *not* affect array subscripting. See Section 4.5 [Case-sensitivity in Matching], page 33.

If `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159), then `IGNORECASE` has no special meaning, and string and regexp operations are always case-sensitive.

`OFMT` This string controls conversion of numbers to strings (see Section 7.4 [Conversion of Strings and Numbers], page 79) for printing with the `print` statement. It works by being passed, in effect, as the first argument to the `sprintf` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135). Its default value is `"%.6g"`. Earlier versions of `awk` also used `OFMT` to specify the format for converting numbers to strings in general expressions; this is now done by `CONVFMT`.

`OFS` This is the output field separator (see Section 6.3 [Output Separators], page 61). It is output between the fields output by a `print` statement. Its default value is `" "`, a string consisting of a single space.

`ORS` This is the output record separator. It is output at the end of every `print` statement. Its default value is `"\n"`. (See Section 6.3 [Output Separators], page 61.)

`RS` This is `awk`'s input record separator. Its default value is a string containing a single newline character, which means that an input record consists of a single line of text. It can also be the null string, in which case records are separated by runs of blank lines, or a regexp, in which case records are separated by matches of the regexp in the input text. (See Section 5.1 [How Input is Split into Records], page 37.)

`SUBSEP` `SUBSEP` is the subscript separator. It has the default value of `"\034"`, and is used to separate the parts of the indices of a multi-dimensional array. Thus, the expression `foo["A", "B"]` really accesses `foo["A\034B"]` (see Section 11.9 [Multi-dimensional Arrays], page 128).

10.2 Built-in Variables that Convey Information

This is an alphabetical list of the variables that are set automatically by `awk` on certain occasions in order to provide information to your program. Those variables that are specific to `gawk` are marked with an asterisk, '`*`'.

`ARGC`
`ARGV` The command-line arguments available to `awk` programs are stored in an array called `ARGV`. `ARGC` is the number of command-line arguments present. See Section 14.2 [Other Command Line Arguments], page 163. Unlike most `awk` arrays, `ARGV` is indexed from zero to `ARGC` − 1. For example:

```
$ awk 'BEGIN {
>         for (i = 0; i < ARGC; i++)
>             print ARGV[i]
>       }' inventory-shipped BBS-list
 ⊣ awk
 ⊣ inventory-shipped
 ⊣ BBS-list
```

In this example, `ARGV[0]` contains `"awk"`, `ARGV[1]` contains `"inventory-shipped"`, and `ARGV[2]` contains `"BBS-list"`. The value of `ARGC` is three, one more than the index of the last element in `ARGV`, since the elements are numbered from zero.

The names `ARGC` and `ARGV`, as well as the convention of indexing the array from zero to `ARGC` − 1, are derived from the C language's method of accessing command line arguments. See Section 10.3 [Using `ARGC` and `ARGV`], page 117, for information about how `awk` uses these variables.

`ARGIND *` The index in `ARGV` of the current file being processed. Every time `gawk` opens a new data file for processing, it sets `ARGIND` to the index in `ARGV` of the file name. When `gawk` is processing the input files, it is always true that '`FILENAME == ARGV[ARGIND]`'.

This variable is useful in file processing; it allows you to tell how far along you are in the list of data files, and to distinguish between successive instances of the same filename on the command line.

While you can change the value of `ARGIND` within your `awk` program, `gawk` will automatically set it to a new value when the next file is opened.

This variable is a `gawk` extension. In other `awk` implementations, or if `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159), it is not special.

`ENVIRON` An associative array that contains the values of the environment. The array indices are the environment variable names; the values are the values of the particular environment variables. For example, `ENVIRON["HOME"]` might be '`/home/arnold`'. Changing this array does not affect the environment passed on to any programs that `awk` may spawn via redirection or the `system` function. (In a future version of `gawk`, it may do so.)

Some operating systems may not have environment variables. On such systems, the `ENVIRON` array is empty (except for `ENVIRON["AWKPATH"]`).

`ERRNO *` If a system error occurs either doing a redirection for `getline`, during a read for `getline`, or during a `close` operation, then `ERRNO` will contain a string describing the error.

This variable is a `gawk` extension. In other `awk` implementations, or if `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159), it is not special.

`FILENAME` This is the name of the file that `awk` is currently reading. When no data files are listed on the command line, `awk` reads from the standard input, and `FILENAME` is set to `"-"`. `FILENAME` is changed each time a new file is read (see Chapter 5 [Reading Input Files], page 37). Inside a `BEGIN` rule, the value of `FILENAME` is `""`, since there are no input files being processed yet.[2] (d.c.)

`FNR` `FNR` is the current record number in the current file. `FNR` is incremented each time a new record is read (see Section 5.8 [Explicit Input with `getline`], page 53). It is reinitialized to zero each time a new input file is started.

`NF` `NF` is the number of fields in the current input record. `NF` is set each time a new record is read, when a new field is created, or when `$0` changes (see Section 5.2 [Examining Fields], page 40).

`NR` This is the number of input records `awk` has processed since the beginning of the program's execution (see Section 5.1 [How Input is Split into Records], page 37). `NR` is set each time a new record is read.

[2] Some early implementations of Unix `awk` initialized `FILENAME` to `"-"`, even if there were data files to be processed. This behavior was incorrect, and should not be relied upon in your programs.

`RLENGTH` `RLENGTH` is the length of the substring matched by the `match` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135). `RLENGTH` is set by invoking the `match` function. Its value is the length of the matched string, or -1 if no match was found.

`RSTART` `RSTART` is the start-index in characters of the substring matched by the `match` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135). `RSTART` is set by invoking the `match` function. Its value is the position of the string where the matched substring starts, or zero if no match was found.

`RT` * `RT` is set each time a record is read. It contains the input text that matched the text denoted by `RS`, the record separator.

This variable is a `gawk` extension. In other `awk` implementations, or if `gawk` is in compatibility mode (see Section 14.1 [Command Line Options], page 159), it is not special.

A side note about `NR` and `FNR`. `awk` simply increments both of these variables each time it reads a record, instead of setting them to the absolute value of the number of records read. This means that your program can change these variables, and their new values will be incremented for each record (d.c.). For example:

```
$ echo '1
> 2
> 3
> 4' | awk 'NR == 2 { NR = 17 }
> { print NR }'
 -| 1
 -| 17
 -| 18
 -| 19
```

Before `FNR` was added to the `awk` language (see Section 17.1 [Major Changes between V7 and SVR3.1], page 247), many `awk` programs used this feature to track the number of records in a file by resetting `NR` to zero when `FILENAME` changed.

10.3 Using `ARGC` and `ARGV`

In Section 10.2 [Built-in Variables that Convey Information], page 115, you saw this program describing the information contained in `ARGC` and `ARGV`:

```
$ awk 'BEGIN {
>         for (i = 0; i < ARGC; i++)
>             print ARGV[i]
>       }' inventory-shipped BBS-list
⊣ awk
⊣ inventory-shipped
⊣ BBS-list
```

In this example, `ARGV[0]` contains `"awk"`, `ARGV[1]` contains `"inventory-shipped"`, and `ARGV[2]` contains `"BBS-list"`.

Notice that the `awk` program is not entered in `ARGV`. The other special command line options, with their arguments, are also not entered. This includes variable assignments done with the '`-v`' option (see Section 14.1 [Command Line Options], page 159). Normal variable assignments on the command line *are* treated as arguments, and do show up in the `ARGV` array.

```
$ cat showargs.awk
⊣ BEGIN {
⊣     printf "A=%d, B=%d\n", A, B
⊣     for (i = 0; i < ARGC; i++)
⊣         printf "\tARGV[%d] = %s\n", i, ARGV[i]
⊣ }
⊣ END   { printf "A=%d, B=%d\n", A, B }
$ awk -v A=1 -f showargs.awk B=2 /dev/null
⊣ A=1, B=0
⊣  ARGV[0] = awk
⊣  ARGV[1] = B=2
⊣  ARGV[2] = /dev/null
⊣ A=1, B=2
```

Your program can alter `ARGC` and the elements of `ARGV`. Each time `awk` reaches the end of an input file, it uses the next element of `ARGV` as the name of the next input file. By storing a different string there, your program can change which files are read. You can use `"-"` to represent the standard input. By storing additional elements and incrementing `ARGC` you can cause additional files to be read.

If you decrease the value of `ARGC`, that eliminates input files from the end of the list. By recording the old value of `ARGC` elsewhere, your program can treat the eliminated arguments as something other than file names.

To eliminate a file from the middle of the list, store the null string (`""`) into `ARGV` in place of the file's name. As a special feature, `awk` ignores file names that have been replaced with the null string. You may also use the `delete` statement to remove elements from `ARGV` (see Section 11.6 [The `delete` Statement], page 125).

All of these actions are typically done from the `BEGIN` rule, before actual processing of the input begins. See Section 16.1.4 [Splitting a Large File Into

Pieces], page 212, and see Section 16.1.5 [Duplicating Output Into Multiple Files], page 214, for an example of each way of removing elements from ARGV.

The following fragment processes ARGV in order to examine, and then remove, command line options.

```
BEGIN {
    for (i = 1; i < ARGC; i++) {
        if (ARGV[i] == "-v")
            verbose = 1
        else if (ARGV[i] == "-d")
            debug = 1
        else if (ARGV[i] ~ /^-?/) {
            e = sprintf("%s: unrecognized option -- %c",
                    ARGV[0], substr(ARGV[i], 1, ,1))
            print e > "/dev/stderr"
        } else
            break
        delete ARGV[i]
    }
}
```

To actually get the options into the awk program, you have to end the awk options with '--', and then supply your options, like so:

```
awk -f myprog -- -v -d file1 file2 ...
```

This is not necessary in gawk: Unless '--posix' has been specified, gawk silently puts any unrecognized options into ARGV for the awk program to deal with.

As soon as it sees an unknown option, gawk stops looking for other options it might otherwise recognize. The above example with gawk would be:

```
gawk -f myprog -d -v file1 file2 ...
```

Since '-d' is not a valid gawk option, the following '-v' is passed on to the awk program.

11 Arrays in awk

An *array* is a table of values, called *elements*. The elements of an array are distinguished by their indices. *Indices* may be either numbers or strings. `awk` maintains a single set of names that may be used for naming variables, arrays and functions (see Chapter 13 [User-defined Functions], page 151). Thus, you cannot have a variable and an array with the same name in the same `awk` program.

11.1 Introduction to Arrays

The `awk` language provides one-dimensional *arrays* for storing groups of related strings or numbers.

Every `awk` array must have a name. Array names have the same syntax as variable names; any valid variable name would also be a valid array name. But you cannot use one name in both ways (as an array and as a variable) in one `awk` program.

Arrays in `awk` superficially resemble arrays in other programming languages; but there are fundamental differences. In `awk`, you don't need to specify the size of an array before you start to use it. Additionally, any number or string in `awk` may be used as an array index, not just consecutive integers.

In most other languages, you have to *declare* an array and specify how many elements or components it contains. In such languages, the declaration causes a contiguous block of memory to be allocated for that many elements. An index in the array usually must be a positive integer; for example, the index zero specifies the first element in the array, which is actually stored at the beginning of the block of memory. Index one specifies the second element, which is stored in memory right after the first element, and so on. It is impossible to add more elements to the array, because it has room for only as many elements as you declared. (Some languages allow arbitrary starting and ending indices, e.g., '`15 .. 27`', but the size of the array is still fixed when the array is declared.)

A contiguous array of four elements might look like this, conceptually, if the element values are eight, `"foo"`, `""` and 30:

8	"foo"	""	30	value
0	1	2	3	index

Only the values are stored; the indices are implicit from the order of the values. Eight is the value at index zero, because eight appears in the position with zero elements before it.

Arrays in `awk` are different: they are *associative*. This means that each array is a collection of pairs: an index, and its corresponding array element value:

```
Element 4       Value 30
Element 2       Value "foo"
Element 1       Value 8
Element 3       Value ""
```

We have shown the pairs in jumbled order because their order is irrelevant.

One advantage of associative arrays is that new pairs can be added at any time. For example, suppose we add to the above array a tenth element whose value is `"number ten"`. The result is this:

```
Element 10      Value "number ten"
Element 4       Value 30
Element 2       Value "foo"
Element 1       Value 8
Element 3       Value ""
```

Now the array is *sparse*, which just means some indices are missing: it has elements 1–4 and 10, but doesn't have elements 5, 6, 7, 8, or 9.

Another consequence of associative arrays is that the indices don't have to be positive integers. Any number, or even a string, can be an index. For example, here is an array which translates words from English into French:

```
Element "dog"   Value "chien"
Element "cat"   Value "chat"
Element "one"   Value "un"
Element 1       Value "un"
```

Here we decided to translate the number one in both spelled-out and numeric form—thus illustrating that a single array can have both numbers and strings as indices. (In fact, array subscripts are always strings; this is discussed in more detail in Section 11.7 [Using Numbers to Subscript Arrays], page 127.)

The value of `IGNORECASE` has no effect upon array subscripting. You must use the exact same string value to retrieve an array element as you used to store it.

When `awk` creates an array for you, e.g., with the `split` built-in function, that array's indices are consecutive integers starting at one. (See Section 12.3 [Built-in Functions for String Manipulation], page 135.)

11.2 Referring to an Array Element

The principal way of using an array is to refer to one of its elements. An array reference is an expression which looks like this:

array`[`*index*`]`

Here, *array* is the name of an array. The expression *index* is the index of the element of the array that you want.

The value of the array reference is the current value of that array element. For example, `foo[4.3]` is an expression for the element of array `foo` at index '`4.3`'.

If you refer to an array element that has no recorded value, the value of the reference is "", the null string. This includes elements to which you have not assigned any value, and elements that have been deleted (see Section 11.6 [The `delete` Statement], page 125). Such a reference automatically creates that array element, with the null string as its value. (In some cases, this is unfortunate, because it might waste memory inside `awk`.)

You can find out if an element exists in an array at a certain index with the expression:

index `in` *array*

This expression tests whether or not the particular index exists, without the side effect of creating that element if it is not present. The expression has the value one (true) if *array*`[`*index*`]` exists, and zero (false) if it does not exist.

For example, to test whether the array `frequencies` contains the index '2', you could write this statement:

```
if (2 in frequencies)
    print "Subscript 2 is present."
```

Note that this is *not* a test of whether or not the array `frequencies` contains an element whose *value* is two. (There is no way to do that except to scan all the elements.) Also, this *does not* create `frequencies[2]`, while the following (incorrect) alternative would do so:

```
if (frequencies[2] != "")
    print "Subscript 2 is present."
```

11.3 Assigning Array Elements

Array elements are lvalues: they can be assigned values just like `awk` variables:

array`[`*subscript*`] = `*value*

Here *array* is the name of your array. The expression *subscript* is the index of the element of the array that you want to assign a value. The expression *value* is the value you are assigning to that element of the array.

11.4 Basic Array Example

The following program takes a list of lines, each beginning with a line number, and prints them out in order of line number. The line numbers are not in order, however, when they are first read: they are scrambled. This program sorts the lines by making an array using the line numbers as subscripts. It then prints out the lines in sorted order of their numbers. It is a very simple program, and gets confused if it encounters repeated numbers, gaps, or lines that don't begin with a number.

```
{
  if ($1 > max)
    max = $1
  arr[$1] = $0
}

END {
  for (x = 1; x <= max; x++)
    print arr[x]
}
```

The first rule keeps track of the largest line number seen so far; it also stores each line into the array `arr`, at an index that is the line's number.

The second rule runs after all the input has been read, to print out all the lines.

When this program is run with the following input:

```
5  I am the Five man
2  Who are you?  The new number two!
4  . . . And four on the floor
1  Who is number one?
3  I three you.
```

its output is this:

```
1  Who is number one?
2  Who are you?  The new number two!
3  I three you.
4  . . . And four on the floor
5  I am the Five man
```

If a line number is repeated, the last line with a given number overrides the others.

Gaps in the line numbers can be handled with an easy improvement to the program's `END` rule:

```
END {
  for (x = 1; x <= max; x++)
    if (x in arr)
      print arr[x]
}
```

11.5 Scanning All Elements of an Array

In programs that use arrays, you often need a loop that executes once for each element of an array. In other languages, where arrays are contiguous and indices are limited to positive integers, this is easy: you can find all the valid indices by counting from the lowest index up to the highest. This technique won't do the job in `awk`, since any number or string can be an

array index. So `awk` has a special kind of `for` statement for scanning an array:

```
for (var in array)
  body
```

This loop executes *body* once for each index in *array* that your program has previously used, with the variable *var* set to that index.

Here is a program that uses this form of the `for` statement. The first rule scans the input records and notes which words appear (at least once) in the input, by storing a one into the array `used` with the word as index. The second rule scans the elements of `used` to find all the distinct words that appear in the input. It prints each word that is more than 10 characters long, and also prints the number of such words. See Section 12.3 [Built-in Functions for String Manipulation], page 135, for more information on the built-in function `length`.

```
# Record a 1 for each word that is used at least once.
{
    for (i = 1; i <= NF; i++)
        used[$i] = 1
}

# Find number of distinct words more than 10 characters long.
END {
    for (x in used)
        if (length(x) > 10) {
            ++num_long_words
            print x
        }
    print num_long_words, "words longer than 10 characters"
}
```

See Section 16.2.5 [Generating Word Usage Counts], page 230, for a more detailed example of this type.

The order in which elements of the array are accessed by this statement is determined by the internal arrangement of the array elements within `awk` and cannot be controlled or changed. This can lead to problems if new elements are added to *array* by statements in the loop body; you cannot predict whether or not the `for` loop will reach them. Similarly, changing *var* inside the loop may produce strange results. It is best to avoid such things.

11.6 The `delete` Statement

You can remove an individual element of an array using the `delete` statement:

```
delete array[index]
```

Once you have deleted an array element, you can no longer obtain any value the element once had. It is as if you had never referred to it and had never given it any value.

Here is an example of deleting elements in an array:

```
for (i in frequencies)
  delete frequencies[i]
```

This example removes all the elements from the array `frequencies`.

If you delete an element, a subsequent `for` statement to scan the array will not report that element, and the `in` operator to check for the presence of that element will return zero (i.e. false):

```
delete foo[4]
if (4 in foo)
    print "This will never be printed"
```

It is important to note that deleting an element is *not* the same as assigning it a null value (the empty string, `""`).

```
foo[4] = ""
if (4 in foo)
  print "This is printed, even though foo[4] is empty"
```

It is not an error to delete an element that does not exist.

You can delete all the elements of an array with a single statement, by leaving off the subscript in the `delete` statement.

```
delete array
```

This ability is a `gawk` extension; it is not available in compatibility mode (see Section 14.1 [Command Line Options], page 159).

Using this version of the `delete` statement is about three times more efficient than the equivalent loop that deletes each element one at a time.

The following statement provides a portable, but non-obvious way to clear out an array.

```
# thanks to Michael Brennan for pointing this out
split("", array)
```

The `split` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135) clears out the target array first. This call asks it to split apart the null string. Since there is no data to split out, the function simply clears the array and then returns.

Caution: Deleting an array does not change its type; you cannot delete an array and then use the array's name as a scalar. For example, this will not work:

```
a[1] = 3; delete a; a = 3
```

11.7 Using Numbers to Subscript Arrays

An important aspect of arrays to remember is that *array subscripts are always strings*. If you use a numeric value as a subscript, it will be converted to a string value before it is used for subscripting (see Section 7.4 [Conversion of Strings and Numbers], page 79).

This means that the value of the built-in variable `CONVFMT` can potentially affect how your program accesses elements of an array. For example:

```
xyz = 12.153
data[xyz] = 1
CONVFMT = "%2.2f"
if (xyz in data)
    printf "%s is in data\n", xyz
else
    printf "%s is not in data\n", xyz
```

This prints '`12.15 is not in data`'. The first statement gives `xyz` a numeric value. Assigning to `data[xyz]` subscripts `data` with the string value `"12.153"` (using the default conversion value of `CONVFMT`, `"%.6g"`), and assigns one to `data["12.153"]`. The program then changes the value of `CONVFMT`. The test '`(xyz in data)`' generates a new string value from `xyz`, this time `"12.15"`, since the value of `CONVFMT` only allows two significant digits. This test fails, since `"12.15"` is a different string from `"12.153"`.

According to the rules for conversions (see Section 7.4 [Conversion of Strings and Numbers], page 79), integer values are always converted to strings as integers, no matter what the value of `CONVFMT` may happen to be. So the usual case of:

```
for (i = 1; i <= maxsub; i++)
    do something with array[i]
```

will work, no matter what the value of `CONVFMT`.

Like many things in `awk`, the majority of the time things work as you would expect them to work. But it is useful to have a precise knowledge of the actual rules, since sometimes they can have a subtle effect on your programs.

11.8 Using Uninitialized Variables as Subscripts

Suppose you want to print your input data in reverse order. A reasonable attempt at a program to do so (with some test data) might look like this:

```
$ echo 'line 1
> line 2
> line 3' | awk '{ l[lines] = $0; ++lines }
> END {
>     for (i = lines-1; i >= 0; --i)
>        print l[i]
> }'
 ⊣ line 3
 ⊣ line 2
```

Unfortunately, the very first line of input data did not come out in the output!

At first glance, this program should have worked. The variable `lines` is uninitialized, and uninitialized variables have the numeric value zero. So, `awk` should have printed the value of `l[0]`.

The issue here is that subscripts for `awk` arrays are **always** strings. And uninitialized variables, when used as strings, have the value `""`, not zero. Thus, '`line 1`' ended up stored in `l[""]`.

The following version of the program works correctly:

```
{ l[lines++] = $0 }
END {
    for (i = lines - 1; i >= 0; --i)
       print l[i]
}
```

Here, the '`++`' forces `lines` to be numeric, thus making the "old value" numeric zero, which is then converted to `"0"` as the array subscript.

As we have just seen, even though it is somewhat unusual, the null string (`""`) is a valid array subscript (d.c.). If '`--lint`' is provided on the command line (see Section 14.1 [Command Line Options], page 159), `gawk` will warn about the use of the null string as a subscript.

11.9 Multi-dimensional Arrays

A multi-dimensional array is an array in which an element is identified by a sequence of indices, instead of a single index. For example, a two-dimensional array requires two indices. The usual way (in most languages, including `awk`) to refer to an element of a two-dimensional array named `grid` is with `grid[`*x*`,`*y*`]`.

Multi-dimensional arrays are supported in `awk` through concatenation of indices into one string. What happens is that `awk` converts the indices into strings (see Section 7.4 [Conversion of Strings and Numbers], page 79) and concatenates them together, with a separator between them. This creates a single string that describes the values of the separate indices. The combined string is used as a single index into an ordinary, one-dimensional array. The separator used is the value of the built-in variable `SUBSEP`.

For example, suppose we evaluate the expression '`foo[5,12] = "value"`' when the value of `SUBSEP` is `"@"`. The numbers five and 12 are converted to strings and concatenated with an '`@`' between them, yielding `"5@12"`; thus, the array element `foo["5@12"]` is set to `"value"`.

Once the element's value is stored, `awk` has no record of whether it was stored with a single index or a sequence of indices. The two expressions '`foo[5,12]`' and '`foo[5 SUBSEP 12]`' are always equivalent.

The default value of `SUBSEP` is the string `"\034"`, which contains a non-printing character that is unlikely to appear in an `awk` program or in most input data.

The usefulness of choosing an unlikely character comes from the fact that index values that contain a string matching `SUBSEP` lead to combined strings that are ambiguous. Suppose that `SUBSEP` were `"@"`; then '`foo["a@b", "c"]`' and '`foo["a", "b@c"]`' would be indistinguishable because both would actually be stored as '`foo["a@b@c"]`'.

You can test whether a particular index-sequence exists in a "multi-dimensional" array with the same operator '`in`' used for single dimensional arrays. Instead of a single index as the left-hand operand, write the whole sequence of indices, separated by commas, in parentheses:

(subscript1, subscript2, ...) `in` *array*

The following example treats its input as a two-dimensional array of fields; it rotates this array 90 degrees clockwise and prints the result. It assumes that all lines have the same number of elements.

```
awk '{
     if (max_nf < NF)
          max_nf = NF
     max_nr = NR
     for (x = 1; x <= NF; x++)
          vector[x, NR] = $x
}

END {
     for (x = 1; x <= max_nf; x++) {
          for (y = max_nr; y >= 1; --y)
               printf("%s ", vector[x, y])
          printf("\n")
     }
}'
```

When given the input:

```
1 2 3 4 5 6
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3
```

it produces:

```
4 3 2 1
5 4 3 2
6 5 4 3
1 6 5 4
2 1 6 5
3 2 1 6
```

11.10 Scanning Multi-dimensional Arrays

There is no special `for` statement for scanning a "multi-dimensional" array; there cannot be one, because in truth there are no multi-dimensional arrays or elements; there is only a multi-dimensional *way of accessing* an array.

However, if your program has an array that is always accessed as multi-dimensional, you can get the effect of scanning it by combining the scanning `for` statement (see Section 11.5 [Scanning All Elements of an Array], page 124) with the `split` built-in function (see Section 12.3 [Built-in Functions for String Manipulation], page 135). It works like this:

```
for (combined in array) {
  split(combined, separate, SUBSEP)
  ...
}
```

This sets `combined` to each concatenated, combined index in the array, and splits it into the individual indices by breaking it apart where the value of `SUBSEP` appears. The split-out indices become the elements of the array `separate`.

Thus, suppose you have previously stored a value in `array[1, "foo"]`; then an element with index `"1\034foo"` exists in `array`. (Recall that the default value of `SUBSEP` is the character with code 034.) Sooner or later the `for` statement will find that index and do an iteration with `combined` set to `"1\034foo"`. Then the `split` function is called as follows:

```
split("1\034foo", separate, "\034")
```

The result of this is to set `separate[1]` to `"1"` and `separate[2]` to `"foo"`. Presto, the original sequence of separate indices has been recovered.

11.11 Using Array Memory Efficiently

This section applies just to `gawk`.

It is often useful to use the same bit of data as an index into multiple arrays. Due to the way `gawk` implements associative arrays, when you need to use input data as an index for multiple arrays, it is much more effecient to assign the input field to a separate variable, and then use that variable as the index.

```
{
        name = $1
        ssn = $2
        nkids = $3
        ...
        seniority[name]++     # better than seniority[$1]++
        kids[name] = nkids    # better than kids[$1] = nkids
}
```

Using separate variables with mnemonic names for the input fields makes programs more readable, in any case. It is an eventual goal to make `gawk`'s array indexing as efficient as possible, no matter what the source of the index value.

12 Built-in Functions

Built-in functions are functions that are always available for your `awk` program to call. This chapter defines all the built-in functions in `awk`; some of them are mentioned in other sections, but they are summarized here for your convenience. (You can also define new functions yourself. See Chapter 13 [User-defined Functions], page 151.)

12.1 Calling Built-in Functions

To call a built-in function, write the name of the function followed by arguments in parentheses. For example, '`atan2(y + z, 1)`' is a call to the function `atan2`, with two arguments.

Whitespace is ignored between the built-in function name and the open-parenthesis, but we recommend that you avoid using whitespace there. User-defined functions do not permit whitespace in this way, and you will find it easier to avoid mistakes by following a simple convention which always works: no whitespace after a function name.

Each built-in function accepts a certain number of arguments. In some cases, arguments can be omitted. The defaults for omitted arguments vary from function to function and are described under the individual functions. In some `awk` implementations, extra arguments given to built-in functions are ignored. However, in `gawk`, it is a fatal error to give extra arguments to a built-in function.

When a function is called, expressions that create the function's actual parameters are evaluated completely before the function call is performed. For example, in the code fragment:

```
i = 4
j = sqrt(i++)
```

the variable `i` is set to five before `sqrt` is called with a value of four for its actual parameter.

The order of evaluation of the expressions used for the function's parameters is undefined. Thus, you should not write programs that assume that parameters are evaluated from left to right or from right to left. For example,

```
i = 5
j = atan2(i++, i *= 2)
```

If the order of evaluation is left to right, then `i` first becomes six, and then 12, and `atan2` is called with the two arguments six and 12. But if the order of evaluation is right to left, `i` first becomes 10, and then 11, and `atan2` is called with the two arguments 11 and 10.

12.2 Numeric Built-in Functions

Here is a full list of built-in functions that work with numbers. Optional parameters are enclosed in square brackets ("[" and "]").

`int(`*x*`)` This produces the nearest integer to *x*, located between *x* and zero, truncated toward zero.

For example, `int(3)` is three, `int(3.9)` is three, `int(-3.9)` is -3, and `int(-3)` is -3 as well.

`sqrt(`*x*`)` This gives you the positive square root of *x*. It reports an error if *x* is negative. Thus, `sqrt(4)` is two.

`exp(`*x*`)` This gives you the exponential of *x* (`e ^ `*x*), or reports an error if *x* is out of range. The range of values *x* can have depends on your machine's floating point representation.

`log(`*x*`)` This gives you the natural logarithm of *x*, if *x* is positive; otherwise, it reports an error.

`sin(`*x*`)` This gives you the sine of *x*, with *x* in radians.

`cos(`*x*`)` This gives you the cosine of *x*, with *x* in radians.

`atan2(`*y*`, `*x*`)`
This gives you the arctangent of *y* `/` *x* in radians.

`rand()` This gives you a random number. The values of `rand` are uniformly-distributed between zero and one. The value is never zero and never one.

Often you want random integers instead. Here is a user-defined function you can use to obtain a random non-negative integer less than *n*:

```
function randint(n) {
     return int(n * rand())
}
```

The multiplication produces a random number greater than zero and less than `n`. We then make it an integer (using `int`) between zero and `n` $-$ 1, inclusive.

Here is an example where a similar function is used to produce random integers between one and *n*. This program prints a new random number for each input record.

```
awk '
# Function to roll a simulated die.
function roll(n) { return 1 + int(rand() * n) }
```

```
# Roll 3 six-sided dice and
# print total number of points.
{
      printf("%d points\n",
             roll(6)+roll(6)+roll(6))
}'
```

Caution: In most `awk` implementations, including `gawk`, `rand` starts generating numbers from the same starting number, or *seed*, each time you run `awk`. Thus, a program will generate the same results each time you run it. The numbers are random within one `awk` run, but predictable from run to run. This is convenient for debugging, but if you want a program to do different things each time it is used, you must change the seed to a value that will be different in each run. To do this, use `srand`.

`srand(`[*x*]`)`

The function `srand` sets the starting point, or seed, for generating random numbers to the value *x*.

Each seed value leads to a particular sequence of random numbers.[1] Thus, if you set the seed to the same value a second time, you will get the same sequence of random numbers again.

If you omit the argument *x*, as in `srand()`, then the current date and time of day are used for a seed. This is the way to get random numbers that are truly unpredictable.

The return value of `srand` is the previous seed. This makes it easy to keep track of the seeds for use in consistently reproducing sequences of random numbers.

12.3 Built-in Functions for String Manipulation

The functions in this section look at or change the text of one or more strings. Optional parameters are enclosed in square brackets ("[" and "]").

`index(`*in*`, `*find*`)`

This searches the string *in* for the first occurrence of the string *find*, and returns the position in characters where that occurrence begins in the string *in*. For example:

```
$ awk 'BEGIN { print index("peanut", "an") }'
⊣ 3
```

If *find* is not found, `index` returns zero. (Remember that string indices in `awk` start at one.)

[1] Computer generated random numbers really are not truly random. They are technically known as "pseudo-random." This means that while the numbers in a sequence appear to be random, you can in fact generate the same sequence of random numbers over and over again.

`length(`[*string*]`)`

This gives you the number of characters in *string*. If *string* is a number, the length of the digit string representing that number is returned. For example, `length("abcde")` is five. By contrast, `length(15 * 35)` works out to three. How? Well, 15 * 35 = 525, and 525 is then converted to the string `"525"`, which has three characters.

If no argument is supplied, `length` returns the length of `$0`.

In older versions of `awk`, you could call the `length` function without any parentheses. Doing so is marked as "deprecated" in the POSIX standard. This means that while you can do this in your programs, it is a feature that can eventually be removed from a future version of the standard. Therefore, for maximal portability of your `awk` programs, you should always supply the parentheses.

`match(`*string*`,` *regexp*`)`

The `match` function searches the string, *string*, for the longest, leftmost substring matched by the regular expression, *regexp*. It returns the character position, or *index*, of where that substring begins (one, if it starts at the beginning of *string*). If no match is found, it returns zero.

The `match` function sets the built-in variable `RSTART` to the index. It also sets the built-in variable `RLENGTH` to the length in characters of the matched substring. If no match is found, `RSTART` is set to zero, and `RLENGTH` to −1.

For example:

```
awk '{
       if ($1 == "FIND")
         regex = $2
       else {
         where = match($0, regex)
         if (where != 0)
           print "Match of", regex, "found at", \
                     where, "in", $0
       }
}'
```

This program looks for lines that match the regular expression stored in the variable `regex`. This regular expression can be changed. If the first word on a line is 'FIND', `regex` is changed to be the second word on that line. Therefore, given:

```
FIND ru+n
My program runs
but not very quickly
```

```
FIND Melvin
JF+KM
This line is property of Reality Engineering Co.
Melvin was here.
```

`awk` prints:

```
Match of ru+n found at 12 in My program runs
Match of Melvin found at 1 in Melvin was here.
```

`split(`*string*`,` *array* [`,` *fieldsep*]`)`

This divides *string* into pieces separated by *fieldsep*, and stores the pieces in *array*. The first piece is stored in *array*`[1]`, the second piece in *array*`[2]`, and so forth. The string value of the third argument, *fieldsep*, is a regexp describing where to split *string* (much as `FS` can be a regexp describing where to split input records). If the *fieldsep* is omitted, the value of `FS` is used. `split` returns the number of elements created.

The `split` function splits strings into pieces in a manner similar to the way input lines are split into fields. For example:

```
split("cul-de-sac", a, "-")
```

splits the string '`cul-de-sac`' into three fields using '`-`' as the separator. It sets the contents of the array `a` as follows:

```
a[1] = "cul"
a[2] = "de"
a[3] = "sac"
```

The value returned by this call to `split` is three.

As with input field-splitting, when the value of *fieldsep* is `" "`, leading and trailing whitespace is ignored, and the elements are separated by runs of whitespace.

Also as with input field-splitting, if *fieldsep* is the null string, each individual character in the string is split into its own array element. (This is a `gawk`-specific extension.)

Recent implementations of `awk`, including `gawk`, allow the third argument to be a regexp constant (`/abc/`), as well as a string (d.c.). The POSIX standard allows this as well.

Before splitting the string, `split` deletes any previously existing elements in the array *array* (d.c.).

If *string* does not match *fieldsep* at all, *array* will have one element. The value of that element will be the original *string*.

`sprintf(`*format*`,` *expression1*`, ...)`

This returns (without printing) the string that `printf` would have printed out with the same arguments (see Section 6.5 [Using `printf` Statements for Fancier Printing], page 62). For example:

```
sprintf("pi = %.2f (approx.)", 22/7)
```

returns the string `"pi = 3.14 (approx.)"`.

`sub(`*regexp*`,` *replacement* `[,` *target*`])`

The `sub` function alters the value of *target*. It searches this value, which is treated as a string, for the leftmost longest substring matched by the regular expression, *regexp*, extending this match as far as possible. Then the entire string is changed by replacing the matched text with *replacement*. The modified string becomes the new value of *target*.

This function is peculiar because *target* is not simply used to compute a value, and not just any expression will do: it must be a variable, field or array element, so that `sub` can store a modified value there. If this argument is omitted, then the default is to use and alter `$0`.

For example:

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

sets `str` to `"wither, water, everywhere"`, by replacing the leftmost, longest occurrence of '`at`' with '`ith`'.

The `sub` function returns the number of substitutions made (either one or zero).

If the special character '`&`' appears in *replacement*, it stands for the precise substring that was matched by *regexp*. (If the regexp can match more than one string, then this precise substring may vary.) For example:

```
awk '{ sub(/candidate/, "& and his wife"); print }'
```

changes the first occurrence of '`candidate`' to '`candidate and his wife`' on each input line.

Here is another example:

```
awk 'BEGIN {
        str = "daabaaa"
        sub(/a+/, "C&C", str)
        print str
}'
 ⊣ dCaaCbaaa
```

This shows how '`&`' can represent a non-constant string, and also illustrates the "leftmost, longest" rule in regexp matching (see Section 4.6 [How Much Text Matches?], page 34).

The effect of this special character ('`&`') can be turned off by putting a backslash before it in the string. As usual, to insert one backslash in the string, you must write two backslashes. Therefore, write '`\\&`' in a string constant to include a literal '`&`'

in the replacement. For example, here is how to replace the first '|' on each line with an '&':

```
awk '{ sub(/\|/, "\\&"); print }'
```

Note: As mentioned above, the third argument to `sub` must be a variable, field or array reference. Some versions of `awk` allow the third argument to be an expression which is not an lvalue. In such a case, `sub` would still search for the pattern and return zero or one, but the result of the substitution (if any) would be thrown away because there is no place to put it. Such versions of `awk` accept expressions like this:

```
sub(/USA/, "United States", "the USA and Canada")
```

For historical compatibility, `gawk` will accept erroneous code, such as in the above example. However, using any other non-changeable object as the third parameter will cause a fatal error, and your program will not run.

Finally, if the *regexp* is not a regexp constant, it is converted into a string and then the value of that string is treated as the regexp to match.

`gsub(`*regexp*`,` *replacement* `[,` *target*`])`

This is similar to the `sub` function, except `gsub` replaces *all* of the longest, leftmost, *non-overlapping* matching substrings it can find. The 'g' in `gsub` stands for "global," which means replace everywhere. For example:

```
awk '{ gsub(/Britain/, "United Kingdom"); print }'
```

replaces all occurrences of the string 'Britain' with 'United Kingdom' for all input records.

The `gsub` function returns the number of substitutions made. If the variable to be searched and altered, *target*, is omitted, then the entire input record, `$0`, is used.

As in `sub`, the characters '&' and '\' are special, and the third argument must be an lvalue.

`gensub(`*regexp*`,` *replacement*`,` *how* `[,` *target*`])`

`gensub` is a general substitution function. Like `sub` and `gsub`, it searches the target string *target* for matches of the regular expression *regexp*. Unlike `sub` and `gsub`, the modified string is returned as the result of the function, and the original target string is *not* changed. If *how* is a string beginning with 'g' or 'G', then it replaces all matches of *regexp* with *replacement*. Otherwise, *how* is a number indicating which match of *regexp* to replace. If no *target* is supplied, `$0` is used instead.

`gensub` provides an additional feature that is not available in `sub` or `gsub`: the ability to specify components of a regexp in the replacement text. This is done by using parentheses in the

regexp to mark the components, and then specifying '*n*' in the replacement text, where *n* is a digit from one to nine. For example:

```
$ gawk '
> BEGIN {
>       a = "abc def"
>       b = gensub(/(.+) (.+)/, "\\2 \\1", "g", a)
>       print b
> }'
 ⊣ def abc
```

As described above for `sub`, you must type two backslashes in order to get one into the string.

In the replacement text, the sequence '`\0`' represents the entire matched text, as does the character '`&`'.

This example shows how you can use the third argument to control which match of the regexp should be changed.

```
$ echo a b c a b c |
> gawk '{ print gensub(/a/, "AA", 2) }'
 ⊣ a b c AA b c
```

In this case, `$0` is used as the default target string. `gensub` returns the new string as its result, which is passed directly to `print` for printing.

If the *how* argument is a string that does not begin with '`g`' or '`G`', or if it is a number that is less than zero, only one substitution is performed.

If *regexp* does not match *target*, `gensub`'s return value is the original, unchanged value of *target*.

`gensub` is a `gawk` extension; it is not available in compatibility mode (see Section 14.1 [Command Line Options], page 159).

`substr(`*string*`,` *start* `[,` *length*`])`

This returns a *length*-character-long substring of *string*, starting at character number *start*. The first character of a string is character number one. For example, `substr("washington", 5, 3)` returns `"ing"`.

If *length* is not present, this function returns the whole suffix of *string* that begins at character number *start*. For example, `substr("washington", 5)` returns `"ington"`. The whole suffix is also returned if *length* is greater than the number of characters remaining in the string, counting from character number *start*.

Note: The string returned by `substr` *cannot* be assigned to. Thus, it is a mistake to attempt to change a portion of a string, like this:

```
string = "abcdef"
# try to get "abCDEf", won't work
substr(string, 3, 3) = "CDE"
```

or to use `substr` as the third agument of `sub` or `gsub`:

```
gsub(/xyz/, "pdq", substr($0, 5, 20))  # WRONG
```

`tolower(`*string*`)`
: This returns a copy of *string*, with each upper-case character in the string replaced with its corresponding lower-case character. Non-alphabetic characters are left unchanged. For example, `tolower("MiXeD cAsE 123")` returns `"mixed case 123"`.

`toupper(`*string*`)`
: This returns a copy of *string*, with each lower-case character in the string replaced with its corresponding upper-case character. Non-alphabetic characters are left unchanged. For example, `toupper("MiXeD cAsE 123")` returns `"MIXED CASE 123"`.

More About '\' and '&' with `sub`, `gsub` and `gensub`

When using `sub`, `gsub` or `gensub`, and trying to get literal backslashes and ampersands into the replacement text, you need to remember that there are several levels of *escape processing* going on.

First, there is the *lexical* level, which is when `awk` reads your program, and builds an internal copy of your program that can be executed.

Then there is the run-time level, when `awk` actually scans the replacement string to determine what to generate.

At both levels, `awk` looks for a defined set of characters that can come after a backslash. At the lexical level, it looks for the escape sequences listed in Section 4.2 [Escape Sequences], page 24. Thus, for every '\' that `awk` will process at the run-time level, you type two '\'s at the lexical level. When a character that is not valid for an escape sequence follows the '\', Unix `awk` and `gawk` both simply remove the initial '\', and put the following character into the string. Thus, for example, `"a\qb"` is treated as `"aqb"`.

At the run-time level, the various functions handle sequences of '\' and '`&`' differently. The situation is (sadly) somewhat complex.

Historically, the `sub` and `gsub` functions treated the two character sequence '`\&`' specially; this sequence was replaced in the generated text with a single '`&`'. Any other '\' within the *replacement* string that did not precede an '`&`' was passed through unchanged. To illustrate with a table:

You type	`sub` sees	`sub` generates
`\&`	`&`	the matched text
`\\&`	`\&`	a literal '`&`'
`\\\&`	`\&`	a literal '`&`'
`\\\\&`	`\\&`	a literal '`\&`'
`\\\\\&`	`\\&`	a literal '`\&`'
`\\\\\\&`	`\\\&`	a literal '`\\&`'
`\\q`	`\q`	a literal '`\q`'

This table shows both the lexical level processing, where an odd number of backslashes becomes an even number at the run time level, and the run-time processing done by `sub`. (For the sake of simplicity, the rest of the tables below only show the case of even numbers of '\'s entered at the lexical level.)

The problem with the historical approach is that there is no way to get a literal '\' followed by the matched text.

The 1992 POSIX standard attempted to fix this problem. The standard says that `sub` and `gsub` look for either a '\' or an '`&`' after the '\'. If either one follows a '\', that character is output literally. The interpretation of '\' and '`&`' then becomes like this:

You type	`sub` sees	`sub` generates
`&`	`&`	the matched text
`\\&`	`\&`	a literal '`&`'
`\\\\&`	`\\&`	a literal '\', then the matched text
`\\\\\\&`	`\\\&`	a literal '`\&`'

This would appear to solve the problem. Unfortunately, the phrasing of the standard is unusual. It says, in effect, that '\' turns off the special meaning of any following character, but that for anything other than '\' and '`&`', such special meaning is undefined. This wording leads to two problems.

1. Backslashes must now be doubled in the *replacement* string, breaking historical `awk` programs.
2. To make sure that an `awk` program is portable, *every* character in the *replacement* string must be preceded with a backslash.[2]

The POSIX standard is under revision.[3] Because of the above problems, proposed text for the revised standard reverts to rules that correspond more closely to the original existing practice. The proposed rules have special cases that make it possible to produce a '\' preceding the matched text.

[2] This consequence was certainly unintended.

[3] As of July, 2000, with final approval and publication as part of the Austin Group Standards hopefully sometime in 2001.

You type	`sub` sees	`sub` generates
`\\\\\\&`	`\\\&`	a literal '`\&`'
`\\\\&`	`\\&`	a literal '`\`', followed by the matched text
`\\&`	`\&`	a literal '`&`'
`\\q`	`\q`	a literal '`\q`'

In a nutshell, at the run-time level, there are now three special sequences of characters, '`\\\&`', '`\\&`' and '`\&`', whereas historically, there was only one. However, as in the historical case, any '`\`' that is not part of one of these three sequences is not special, and appears in the output literally.

`gawk` 3.0 follows these proposed POSIX rules for `sub` and `gsub`. Whether these proposed rules will actually become codified into the standard is unknown at this point. Subsequent `gawk` releases will track the standard and implement whatever the final version specifies; this book will be updated as well.

The rules for `gensub` are considerably simpler. At the run-time level, whenever `gawk` sees a '`\`', if the following character is a digit, then the text that matched the corresponding parenthesized subexpression is placed in the generated output. Otherwise, no matter what the character after the '`\`' is, that character will appear in the generated text, and the '`\`' will not.

You type	`gensub` sees	`gensub` generates
`&`	`&`	the matched text
`\\&`	`\&`	a literal '`&`'
`\\\\`	`\\`	a literal '`\`'
`\\\\&`	`\\&`	a literal '`\`', then the matched text
`\\\\\\&`	`\\\&`	a literal '`\&`'
`\\q`	`\q`	a literal '`q`'

Because of the complexity of the lexical and run-time level processing, and the special cases for `sub` and `gsub`, we recommend the use of `gawk` and `gensub` for when you have to do substitutions.

12.4 Built-in Functions for Input/Output

The following functions are related to Input/Output (I/O). Optional parameters are enclosed in square brackets ("[" and "]").

`close(`*filename*`)`
: Close the file *filename*, for input or output. The argument may alternatively be a shell command that was used for redirecting to or from a pipe; then the pipe is closed. See Section 6.8

[Closing Input and Output Files and Pipes], page 71, for more information.

`fflush(`[*filename*]`)`

Flush any buffered output associated *filename*, which is either a file opened for writing, or a shell command for redirecting output to a pipe.

Many utility programs will *buffer* their output; they save information to be written to a disk file or terminal in memory, until there is enough for it to be worthwhile to send the data to the ouput device. This is often more efficient than writing every little bit of information as soon as it is ready. However, sometimes it is necessary to force a program to *flush* its buffers; that is, write the information to its destination, even if a buffer is not full. This is the purpose of the `fflush` function; `gawk` too buffers its output, and the `fflush` function can be used to force `gawk` to flush its buffers.

`fflush` is a recent (1994) addition to the Bell Labs research version of `awk`; it is not part of the POSIX standard, and will not be available if '`--posix`' has been specified on the command line (see Section 14.1 [Command Line Options], page 159).

`gawk` extends the `fflush` function in two ways. The first is to allow no argument at all. In this case, the buffer for the standard output is flushed. The second way is to allow the null string (`""`) as the argument. In this case, the buffers for *all* open output files and pipes are flushed.

`fflush` returns zero if the buffer was successfully flushed, and nonzero otherwise.

`system(`*command*`)`

The `system` function allows the user to execute operating system commands and then return to the `awk` program. The `system` function executes the command given by the string *command*. It returns, as its value, the status returned by the command that was executed.

For example, if the following fragment of code is put in your `awk` program:

```
END {
     system("date | mail -s 'awk run done' root")
}
```

the system administrator will be sent mail when the `awk` program finishes processing input and begins its end-of-input processing.

Note that redirecting `print` or `printf` into a pipe is often enough to accomplish your task. If you need to run many commands, it will be more efficient to simply print them to a pipe to the shell:

```
while (more stuff to do)
    print command | "/bin/sh"
close("/bin/sh")
```

However, if your `awk` program is interactive, `system` is useful for cranking up large self-contained programs, such as a shell or an editor.

Some operating systems cannot implement the `system` function. `system` causes a fatal error if it is not supported.

Interactive vs. Non-Interactive Buffering

As a side point, buffering issues can be even more confusing depending upon whether or not your program is *interactive*, i.e., communicating with a user sitting at a keyboard.[4]

Interactive programs generally *line buffer* their output; they write out every line. Non-interactive programs wait until they have a full buffer, which may be many lines of output.

Here is an example of the difference.

```
$ awk '{ print $1 + $2 }'
1 1
⊣ 2
2 3
⊣ 5
Control-d
```

Each line of output is printed immediately. Compare that behavior with this example.

```
$ awk '{ print $1 + $2 }' | cat
1 1
2 3
Control-d
⊣ 2
⊣ 5
```

Here, no output is printed until after the *Control-d* is typed, since it is all buffered, and sent down the pipe to `cat` in one shot.

Controlling Output Buffering with `system`

The `fflush` function provides explicit control over output buffering for individual files and pipes. However, its use is not portable to many other `awk` implementations. An alternative method to flush output buffers is by calling `system` with a null string as its argument:

[4] A program is interactive if the standard output is connected to a terminal device.

```
system("")   # flush output
```

`gawk` treats this use of the `system` function as a special case, and is smart enough not to run a shell (or other command interpreter) with the empty command. Therefore, with `gawk`, this idiom is not only useful, it is efficient. While this method should work with other `awk` implementations, it will not necessarily avoid starting an unnecessary shell. (Other implementations may only flush the buffer associated with the standard output, and not necessarily all buffered output.)

If you think about what a programmer expects, it makes sense that `system` should flush any pending output. The following program:

```
BEGIN {
     print "first print"
     system("echo system echo")
     print "second print"
}
```

must print

```
first print
system echo
second print
```

and not

```
system echo
first print
second print
```

If `awk` did not flush its buffers before calling `system`, the latter (undesirable) output is what you would see.

12.5 Functions for Dealing with Time Stamps

A common use for `awk` programs is the processing of log files containing time stamp information, indicating when a particular log record was written. Many programs log their time stamp in the form returned by the `time` system call, which is the number of seconds since a particular epoch. On POSIX systems, it is the number of seconds since Midnight, January 1, 1970, UTC.

In order to make it easier to process such log files, and to produce useful reports, `gawk` provides two functions for working with time stamps. Both of these are `gawk` extensions; they are not specified in the POSIX standard, nor are they in any other known version of `awk`.

Optional parameters are enclosed in square brackets ("[" and "]").

`systime()`
: This function returns the current time as the number of seconds since the system epoch. On POSIX systems, this is the number of seconds since Midnight, January 1, 1970, UTC. It may be a different number on other systems.

`strftime`([*format* [`,` *timestamp*]])
: This function returns a string. It is similar to the function of the same name in ANSI C. The time specified by *timestamp* is used to produce a string, based on the contents of the *format* string. The *timestamp* is in the same format as the value returned by the `systime` function. If no *timestamp* argument is supplied, `gawk` will use the current time of day as the time stamp. If no *format* argument is supplied, `strftime` uses `"%a %b %d %H:%M:%S %Z %Y"`. This format string produces output (almost) equivalent to that of the `date` utility. (Versions of `gawk` prior to 3.0 require the *format* argument.)

The `systime` function allows you to compare a time stamp from a log file with the current time of day. In particular, it is easy to determine how long ago a particular record was logged. It also allows you to produce log records using the "seconds since the epoch" format.

The `strftime` function allows you to easily turn a time stamp into human-readable information. It is similar in nature to the `sprintf` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135), in that it copies non-format specification characters verbatim to the returned string, while substituting date and time values for format specifications in the *format* string.

`strftime` is guaranteed by the ANSI C standard to support the following date format specifications:

`%a`
: The locale's abbreviated weekday name.

`%A`
: The locale's full weekday name.

`%b`
: The locale's abbreviated month name.

`%B`
: The locale's full month name.

`%c`
: The locale's "appropriate" date and time representation.

`%d`
: The day of the month as a decimal number (01–31).

`%H`
: The hour (24-hour clock) as a decimal number (00–23).

`%I`
: The hour (12-hour clock) as a decimal number (01–12).

`%j`
: The day of the year as a decimal number (001–366).

`%m`
: The month as a decimal number (01–12).

`%M`
: The minute as a decimal number (00–59).

`%p`
: The locale's equivalent of the AM/PM designations associated with a 12-hour clock.

`%S`
: The second as a decimal number (00–60).[5]

[5] Occasionally there are minutes in a year with a leap second, which is why the seconds can go up to 60.

`%U` The week number of the year (the first Sunday as the first day of week one) as a decimal number (00–53).

`%w` The weekday as a decimal number (0–6). Sunday is day zero.

`%W` The week number of the year (the first Monday as the first day of week one) as a decimal number (00–53).

`%x` The locale's "appropriate" date representation.

`%X` The locale's "appropriate" time representation.

`%y` The year without century as a decimal number (00–99).

`%Y` The year with century as a decimal number (e.g., 1995).

`%Z` The time zone name or abbreviation, or no characters if no time zone is determinable.

`%%` A literal '`%`'.

If a conversion specifier is not one of the above, the behavior is undefined.[6]

Informally, a *locale* is the geographic place in which a program is meant to run. For example, a common way to abbreviate the date September 4, 1991 in the United States would be "9/4/91". In many countries in Europe, however, it would be abbreviated "4.9.91". Thus, the '`%x`' specification in a `"US"` locale might produce '`9/4/91`', while in a `"EUROPE"` locale, it might produce '`4.9.91`'. The ANSI C standard defines a default `"C"` locale, which is an environment that is typical of what most C programmers are used to.

A public-domain C version of `strftime` is supplied with `gawk` for systems that are not yet fully ANSI-compliant. If that version is used to compile `gawk` (see Appendix B [Installing `gawk`], page 273), then the following additional format specifications are available:

`%D` Equivalent to specifying '`%m/%d/%y`'.

`%e` The day of the month, padded with a space if it is only one digit.

`%h` Equivalent to '`%b`', above.

`%n` A newline character (ASCII LF).

`%r` Equivalent to specifying '`%I:%M:%S %p`'.

`%R` Equivalent to specifying '`%H:%M`'.

`%T` Equivalent to specifying '`%H:%M:%S`'.

`%t` A tab character.

`%k` The hour (24-hour clock) as a decimal number (0-23). Single digit numbers are padded with a space.

[6] This is because ANSI C leaves the behavior of the C version of `strftime` undefined, and `gawk` will use the system's version of `strftime` if it's there. Typically, the conversion specifier will either not appear in the returned string, or it will appear literally.

`%l` The hour (12-hour clock) as a decimal number (1-12). Single digit numbers are padded with a space.

`%C` The century, as a number between 00 and 99.

`%u` The weekday as a decimal number [1 (Monday)–7].

`%V` The week number of the year (the first Monday as the first day of week one) as a decimal number (01–53). The method for determining the week number is as specified by ISO 8601 (to wit: if the week containing January 1 has four or more days in the new year, then it is week one, otherwise it is week 53 of the previous year and the next week is week one).

`%G` The year with century of the ISO week number, as a decimal number.

For example, January 1, 1993, is in week 53 of 1992. Thus, the year of its ISO week number is 1992, even though its year is 1993. Similarly, December 31, 1973, is in week 1 of 1974. Thus, the year of its ISO week number is 1974, even though its year is 1973.

`%g` The year without century of the ISO week number, as a decimal number (00–99).

`%Ec %EC %Ex %Ey %EY %Od %Oe %OH %OI`
`%Om %OM %OS %Ou %OU %OV %Ow %OW %Oy`
These are "alternate representations" for the specifications that use only the second letter ('`%c`', '`%C`', and so on). They are recognized, but their normal representations are used.[7] (These facilitate compliance with the POSIX `date` utility.)

`%v` The date in VMS format (e.g., 20-JUN-1991).

`%z` The timezone offset in a +HHMM format (e.g., the format necessary to produce RFC-822/RFC-1036 date headers).

This example is an `awk` implementation of the POSIX `date` utility. Normally, the `date` utility prints the current date and time of day in a well known format. However, if you provide an argument to it that begins with a '`+`', `date` will copy non-format specifier characters to the standard output, and will interpret the current time according to the format specifiers in the string. For example:

```
$ date '+Today is %A, %B %d, %Y.'
⊣ Today is Thursday, July 11, 1991.
```

Here is the `gawk` version of the `date` utility. It has a shell "wrapper", to handle the '`-u`' option, which requires that `date` run as if the time zone was set to UTC.

[7] If you don't understand any of this, don't worry about it; these facilities are meant to make it easier to "internationalize" programs.

```
#! /bin/sh
#
# date --- approximate the P1003.2 'date' command

case $1 in
-u)  TZ=GMT0     # use UTC
     export TZ
     shift ;;
esac

gawk 'BEGIN  {
    format = "%a %b %d %H:%M:%S %Z %Y"
    exitval = 0

    if (ARGC > 2)
        exitval = 1
    else if (ARGC == 2) {
        format = ARGV[1]
        if (format ~ /^\+/)
            format = substr(format, 2)   # remove leading +
    }
    print strftime(format)
    exit exitval
}' "$@"
```

13 User-defined Functions

Complicated `awk` programs can often be simplified by defining your own functions. User-defined functions can be called just like built-in ones (see Section 7.13 [Function Calls], page 91), but it is up to you to define them—to tell `awk` what they should do.

13.1 Function Definition Syntax

Definitions of functions can appear anywhere between the rules of an `awk` program. Thus, the general form of an `awk` program is extended to include sequences of rules *and* user-defined function definitions. There is no need in `awk` to put the definition of a function before all uses of the function. This is because `awk` reads the entire program before starting to execute any of it.

The definition of a function named *name* looks like this:

```
function name(parameter-list)
{
      body-of-function
}
```

name is the name of the function to be defined. A valid function name is like a valid variable name: a sequence of letters, digits and underscores, not starting with a digit. Within a single `awk` program, any particular name can only be used as a variable, array or function.

parameter-list is a list of the function's arguments and local variable names, separated by commas. When the function is called, the argument names are used to hold the argument values given in the call. The local variables are initialized to the empty string. A function cannot have two parameters with the same name.

The *body-of-function* consists of `awk` statements. It is the most important part of the definition, because it says what the function should actually *do*. The argument names exist to give the body a way to talk about the arguments; local variables, to give the body places to keep temporary values.

Argument names are not distinguished syntactically from local variable names; instead, the number of arguments supplied when the function is called determines how many argument variables there are. Thus, if three argument values are given, the first three names in *parameter-list* are arguments, and the rest are local variables.

It follows that if the number of arguments is not the same in all calls to the function, some of the names in *parameter-list* may be arguments on some occasions and local variables on others. Another way to think of this is that omitted arguments default to the null string.

Usually when you write a function you know how many names you intend to use for arguments and how many you intend to use as local variables. It

is conventional to place some extra space between the arguments and the local variables, to document how your function is supposed to be used.

During execution of the function body, the arguments and local variable values hide or *shadow* any variables of the same names used in the rest of the program. The shadowed variables are not accessible in the function definition, because there is no way to name them while their names have been taken away for the local variables. All other variables used in the `awk` program can be referenced or set normally in the function's body.

The arguments and local variables last only as long as the function body is executing. Once the body finishes, you can once again access the variables that were shadowed while the function was running.

The function body can contain expressions which call functions. They can even call this function, either directly or by way of another function. When this happens, we say the function is *recursive*.

In many `awk` implementations, including `gawk`, the keyword `function` may be abbreviated `func`. However, POSIX only specifies the use of the keyword `function`. This actually has some practical implications. If `gawk` is in POSIX-compatibility mode (see Section 14.1 [Command Line Options], page 159), then the following statement will *not* define a function:

```
func foo() { a = sqrt($1) ; print a }
```

Instead it defines a rule that, for each record, concatenates the value of the variable '`func`' with the return value of the function '`foo`'. If the resulting string is non-null, the action is executed. This is probably not what was desired. (`awk` accepts this input as syntactically valid, since functions may be used before they are defined in `awk` programs.)

To ensure that your `awk` programs are portable, always use the keyword `function` when defining a function.

13.2 Function Definition Examples

Here is an example of a user-defined function, called `myprint`, that takes a number and prints it in a specific format.

```
function myprint(num)
{
     printf "%6.3g\n", num
}
```

To illustrate, here is an `awk` rule which uses our `myprint` function:

```
$3 > 0     { myprint($3) }
```

This program prints, in our special format, all the third fields that contain a positive number in our input. Therefore, when given:

```
 1.2   3.4    5.6   7.8
 9.10 11.12 -13.14 15.16
17.18 19.20  21.22 23.24
```

this program, using our function to format the results, prints:

```
   5.6
  21.2
```

This function deletes all the elements in an array.

```
function delarray(a,    i)
{
    for (i in a)
       delete a[i]
}
```

When working with arrays, it is often necessary to delete all the elements in an array and start over with a new list of elements (see Section 11.6 [The `delete` Statement], page 125). Instead of having to repeat this loop everywhere in your program that you need to clear out an array, your program can just call `delarray`. (This guarantees portability. The usage '`delete` *array*' to delete the contents of an entire array is a non-standard extension.)

Here is an example of a recursive function. It takes a string as an input parameter, and returns the string in backwards order.

```
function rev(str, start)
{
    if (start == 0)
        return ""

    return (substr(str, start, 1) rev(str, start - 1))
}
```

If this function is in a file named '`rev.awk`', we can test it this way:

```
$ echo "Don't Panic!" |
> gawk --source '{ print rev($0, length($0)) }' -f rev.awk
 ⊣ !cinaP t'noD
```

Here is an example that uses the built-in function `strftime`. (See Section 12.5 [Functions for Dealing with Time Stamps], page 146, for more information on `strftime`.) The C `ctime` function takes a timestamp and returns it in a string, formatted in a well known fashion. Here is an `awk` version:

```
# ctime.awk
#
# awk version of C ctime(3) function
```

```
function ctime(ts,    format)
{
    format = "%a %b %d %H:%M:%S %Z %Y"
    if (ts == 0)
        ts = systime()       # use current time as default
    return strftime(format, ts)
}
```

13.3 Calling User-defined Functions

Calling a function means causing the function to run and do its job. A function call is an expression, and its value is the value returned by the function.

A function call consists of the function name followed by the arguments in parentheses. What you write in the call for the arguments are `awk` expressions; each time the call is executed, these expressions are evaluated, and the values are the actual arguments. For example, here is a call to `foo` with three arguments (the first being a string concatenation):

```
foo(x y, "lose", 4 * z)
```

Caution: whitespace characters (spaces and tabs) are not allowed between the function name and the open-parenthesis of the argument list. If you write whitespace by mistake, `awk` might think that you mean to concatenate a variable with an expression in parentheses. However, it notices that you used a function name and not a variable name, and reports an error.

When a function is called, it is given a *copy* of the values of its arguments. This is known as *call by value*. The caller may use a variable as the expression for the argument, but the called function does not know this: it only knows what value the argument had. For example, if you write this code:

```
foo = "bar"
z = myfunc(foo)
```

then you should not think of the argument to `myfunc` as being "the variable `foo`." Instead, think of the argument as the string value, `"bar"`.

If the function `myfunc` alters the values of its local variables, this has no effect on any other variables. Thus, if `myfunc` does this:

```
function myfunc(str)
{
  print str
  str = "zzz"
  print str
}
```

to change its first argument variable `str`, this *does not* change the value of `foo` in the caller. The role of `foo` in calling `myfunc` ended when its value, `"bar"`, was computed. If `str` also exists outside of `myfunc`, the function body

cannot alter this outer value, because it is shadowed during the execution of `myfunc` and cannot be seen or changed from there.

However, when arrays are the parameters to functions, they are *not* copied. Instead, the array itself is made available for direct manipulation by the function. This is usually called *call by reference*. Changes made to an array parameter inside the body of a function *are* visible outside that function. *This can be very dangerous if you do not watch what you are doing.* For example:

```
function changeit(array, ind, nvalue)
{
     array[ind] = nvalue
}

BEGIN {
    a[1] = 1; a[2] = 2; a[3] = 3
    changeit(a, 2, "two")
    printf "a[1] = %s, a[2] = %s, a[3] = %s\n",
            a[1], a[2], a[3]
}
```

This program prints '`a[1] = 1, a[2] = two, a[3] = 3`', because `changeit` stores `"two"` in the second element of `a`.

Some `awk` implementations allow you to call a function that has not been defined, and only report a problem at run-time when the program actually tries to call the function. For example:

```
BEGIN {
    if (0)
        foo()
    else
        bar()
}
function bar() { ... }
# note that 'foo' is not defined
```

Since the '`if`' statement will never be true, it is not really a problem that `foo` has not been defined. Usually though, it is a problem if a program calls an undefined function.

If '`--lint`' has been specified (see Section 14.1 [Command Line Options], page 159), `gawk` will report about calls to undefined functions.

Some `awk` implementations generate a run-time error if you use the `next` statement (see Section 9.7 [The `next` Statement], page 108) inside a user-defined function. `gawk` does not have this problem.

13.4 The `return` Statement

The body of a user-defined function can contain a `return` statement. This statement returns control to the rest of the `awk` program. It can also be used to return a value for use in the rest of the `awk` program. It looks like this:

`return` [*expression*]

The *expression* part is optional. If it is omitted, then the returned value is undefined and, therefore, unpredictable.

A `return` statement with no value expression is assumed at the end of every function definition. So if control reaches the end of the function body, then the function returns an unpredictable value. `awk` will *not* warn you if you use the return value of such a function.

Sometimes, you want to write a function for what it does, not for what it returns. Such a function corresponds to a `void` function in C or to a `procedure` in Pascal. Thus, it may be appropriate to not return any value; you should simply bear in mind that if you use the return value of such a function, you do so at your own risk.

Here is an example of a user-defined function that returns a value for the largest number among the elements of an array:

```
function maxelt(vec,   i, ret)
{
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
               ret = vec[i]
     }
     return ret
}
```

You call `maxelt` with one argument, which is an array name. The local variables `i` and `ret` are not intended to be arguments; while there is nothing to stop you from passing two or three arguments to `maxelt`, the results would be strange. The extra space before `i` in the function parameter list indicates that `i` and `ret` are not supposed to be arguments. This is a convention that you should follow when you define functions.

Here is a program that uses our `maxelt` function. It loads an array, calls `maxelt`, and then reports the maximum number in that array:

```
awk '
function maxelt(vec,   i, ret)
{
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
               ret = vec[i]
     }
     return ret
}

# Load all fields of each record into nums.
{
     for(i = 1; i <= NF; i++)
          nums[NR, i] = $i
}

END {
     print maxelt(nums)
}'
```

Given the following input:

```
 1 5 23 8 16
44 3 5 2 8 26
256 291 1396 2962 100
-6 467 998 1101
99385 11 0 225
```

our program tells us (predictably) that `99385` is the largest number in our array.

14 Running awk

There are two ways to run `awk`: with an explicit program, or with one or more program files. Here are templates for both of them; items enclosed in '[...]' in these templates are optional.

Besides traditional one-letter POSIX-style options, `gawk` also supports GNU long options.

```
awk [options] -f progfile [--] file ...
awk [options] [--] 'program' file ...
```

It is possible to invoke `awk` with an empty program:

```
$ awk '' datafile1 datafile2
```

Doing so makes little sense though; `awk` will simply exit silently when given an empty program (d.c.). If '`--lint`' has been specified on the command line, `gawk` will issue a warning that the program is empty.

14.1 Command Line Options

Options begin with a dash, and consist of a single character. GNU style long options consist of two dashes and a keyword. The keyword can be abbreviated, as long the abbreviation allows the option to be uniquely identified. If the option takes an argument, then the keyword is either immediately followed by an equals sign ('`=`') and the argument's value, or the keyword and the argument's value are separated by whitespace. For brevity, the discussion below only refers to the traditional short options; however the long and short options are interchangeable in all contexts.

Each long option for `gawk` has a corresponding POSIX-style option. The options and their meanings are as follows:

`-F` *fs*
`--field-separator` *fs*
: Sets the `FS` variable to *fs* (see Section 5.5 [Specifying How Fields are Separated], page 44).

`-f` *source-file*
`--file` *source-file*
: Indicates that the `awk` program is to be found in *source-file* instead of in the first non-option argument.

`-v` *var*`=`*val*
`--assign` *var*`=`*val*
: Sets the variable *var* to the value *val* **before** execution of the program begins. Such variable values are available inside the `BEGIN` rule (see Section 14.2 [Other Command Line Arguments], page 163).

The '`-v`' option can only set one variable, but you can use it more than once, setting another variable each time, like this: '`awk -v foo=1 -v bar=2 ...`'.

Caution: Using '`-v`' to set the values of the builtin variables may lead to suprising results. `awk` will reset the values of those variables as it needs to, possibly ignoring any predefined value you may have given.

`-mf` *NNN*
`-mr` *NNN* Set various memory limits to the value *NNN*. The '`f`' flag sets the maximum number of fields, and the '`r`' flag sets the maximum record size. These two flags and the '`-m`' option are from the Bell Labs research version of Unix `awk`. They are provided for compatibility, but otherwise ignored by `gawk`, since `gawk` has no predefined limits.

`-W` *gawk-opt*
Following the POSIX standard, options that are implementation specific are supplied as arguments to the '`-W`' option. These options also have corresponding GNU style long options. See below.

`--` Signals the end of the command line options. The following arguments are not treated as options even if they begin with '`-`'. This interpretation of '`--`' follows the POSIX argument parsing conventions.

This is useful if you have file names that start with '`-`', or in shell scripts, if you have file names that will be specified by the user which could start with '`-`'.

The following `gawk`-specific options are available:

`-W traditional`
`-W compat`
`--traditional`
`--compat` Specifies *compatibility mode*, in which the GNU extensions to the `awk` language are disabled, so that `gawk` behaves just like the Bell Labs research version of Unix `awk`. '`--traditional`' is the preferred form of this option. See Section 17.5 [Extensions in `gawk` Not in POSIX `awk`], page 249, which summarizes the extensions. Also see Section C.1 [Downward Compatibility and Debugging], page 289.

`-W copyleft`
`-W copyright`
`--copyleft`
`--copyright`
Print the short version of the General Public License, and then exit. This option may disappear in a future version of `gawk`.

`-W help`
`-W usage`
`--help`
`--usage` Print a "usage" message summarizing the short and long style options that `gawk` accepts, and then exit.

`-W lint`
`--lint` Warn about constructs that are dubious or non-portable to other `awk` implementations. Some warnings are issued when `gawk` first reads your program. Others are issued at run-time, as your program executes.

`-W lint-old`
`--lint-old`
Warn about constructs that are not available in the original Version 7 Unix version of `awk` (see Section 17.1 [Major Changes between V7 and SVR3.1], page 247).

`-W posix`
`--posix` Operate in strict POSIX mode. This disables all `gawk` extensions (just like '`--traditional`'), and adds the following additional restrictions:

- `\x` escape sequences are not recognized (see Section 4.2 [Escape Sequences], page 24).
- Newlines do not act as whitespace to separate fields when `FS` is equal to a single space.
- The synonym `func` for the keyword `function` is not recognized (see Section 13.1 [Function Definition Syntax], page 151).
- The operators '`**`' and '`**=`' cannot be used in place of '`^`' and '`^=`' (see Section 7.5 [Arithmetic Operators], page 80, and also see Section 7.7 [Assignment Expressions], page 82).
- Specifying '`-Ft`' on the command line does not set the value of `FS` to be a single tab character (see Section 5.5 [Specifying How Fields are Separated], page 44).
- The `fflush` built-in function is not supported (see Section 12.4 [Built-in Functions for Input/Output], page 143).

If you supply both '`--traditional`' and '`--posix`' on the command line, '`--posix`' will take precedence. `gawk` will also issue a warning if both options are supplied.

`-W re-interval`
`--re-interval`
Allow interval expressions (see Section 4.3 [Regular Expression Operators], page 26), in regexps. Because interval expressions were traditionally not available in `awk`, `gawk` does not provide them by default. This prevents old `awk` programs from breaking.

`-W source` *program-text*
`--source` *program-text*
: Program source code is taken from the *program-text*. This option allows you to mix source code in files with source code that you enter on the command line. This is particularly useful when you have library functions that you wish to use from your command line programs (see Section 14.3 [The `AWKPATH` Environment Variable], page 164).

`-W version`
`--version`
: Prints version information for this particular copy of `gawk`. This allows you to determine if your copy of `gawk` is up to date with respect to whatever the Free Software Foundation is currently distributing. It is also useful for bug reports (see Section B.7 [Reporting Problems and Bugs], page 285).

Any other options are flagged as invalid with a warning message, but are otherwise ignored.

In compatibility mode, as a special case, if the value of *fs* supplied to the '`-F`' option is '`t`', then `FS` is set to the tab character (`"\t"`). This is only true for '`--traditional`', and not for '`--posix`' (see Section 5.5 [Specifying How Fields are Separated], page 44).

The '`-f`' option may be used more than once on the command line. If it is, `awk` reads its program source from all of the named files, as if they had been concatenated together into one big file. This is useful for creating libraries of `awk` functions. Useful functions can be written once, and then retrieved from a standard place, instead of having to be included into each individual program.

You can type in a program at the terminal and still use library functions, by specifying '`-f /dev/tty`'. `awk` will read a file from the terminal to use as part of the `awk` program. After typing your program, type *`Control-d`* (the end-of-file character) to terminate it. (You may also use '`-f -`' to read program source from the standard input, but then you will not be able to also use the standard input as a source of data.)

Because it is clumsy using the standard `awk` mechanisms to mix source file and command line `awk` programs, `gawk` provides the '`--source`' option. This does not require you to pre-empt the standard input for your source code, and allows you to easily mix command line and library source code (see Section 14.3 [The `AWKPATH` Environment Variable], page 164).

If no '`-f`' or '`--source`' option is specified, then `gawk` will use the first non-option command line argument as the text of the program source code.

If the environment variable `POSIXLY_CORRECT` exists, then `gawk` will behave in strict POSIX mode, exactly as if you had supplied the '`--posix`' command line option. Many GNU programs look for this environment variable to turn on strict POSIX mode. If you supply '`--lint`' on the command

line, and gawk turns on POSIX mode because of POSIXLY_CORRECT, then it will print a warning message indicating that POSIX mode is in effect.

You would typically set this variable in your shell's startup file. For a Bourne compatible shell (such as Bash), you would add these lines to the '.profile' file in your home directory.

```
POSIXLY_CORRECT=true
export POSIXLY_CORRECT
```

For a csh compatible shell,[1] you would add this line to the '.login' file in your home directory.

```
setenv POSIXLY_CORRECT true
```

14.2 Other Command Line Arguments

Any additional arguments on the command line are normally treated as input files to be processed in the order specified. However, an argument that has the form *var=value*, assigns the value *value* to the variable *var*—it does not specify a file at all.

All these arguments are made available to your awk program in the ARGV array (see Chapter 10 [Built-in Variables], page 113). Command line options and the program text (if present) are omitted from ARGV. All other arguments, including variable assignments, are included. As each element of ARGV is processed, gawk sets the variable ARGIND to the index in ARGV of the current element.

The distinction between file name arguments and variable-assignment arguments is made when awk is about to open the next input file. At that point in execution, it checks the "file name" to see whether it is really a variable assignment; if so, awk sets the variable instead of reading a file.

Therefore, the variables actually receive the given values after all previously specified files have been read. In particular, the values of variables assigned in this fashion are *not* available inside a BEGIN rule (see Section 8.1.5 [The BEGIN and END Special Patterns], page 98), since such rules are run before awk begins scanning the argument list.

The variable values given on the command line are processed for escape sequences (d.c.) (see Section 4.2 [Escape Sequences], page 24).

In some earlier implementations of awk, when a variable assignment occurred before any file names, the assignment would happen *before* the BEGIN rule was executed. awk's behavior was thus inconsistent; some command line assignments were available inside the BEGIN rule, while others were not. However, some applications came to depend upon this "feature." When awk was changed to be more consistent, the '-v' option was added to accommodate applications that depended upon the old behavior.

[1] Not recommended.

The variable assignment feature is most useful for assigning to variables such as `RS`, `OFS`, and `ORS`, which control input and output formats, before scanning the data files. It is also useful for controlling state if multiple passes are needed over a data file. For example:

```
awk 'pass == 1  { pass 1 stuff }
     pass == 2  { pass 2 stuff }' pass=1 mydata pass=2 mydata
```

Given the variable assignment feature, the '`-F`' option for setting the value of `FS` is not strictly necessary. It remains for historical compatibility.

14.3 The `AWKPATH` Environment Variable

The previous section described how `awk` program files can be named on the command line with the '`-f`' option. In most `awk` implementations, you must supply a precise path name for each program file, unless the file is in the current directory.

But in `gawk`, if the file name supplied to the '`-f`' option does not contain a '`/`', then `gawk` searches a list of directories (called the *search path*), one by one, looking for a file with the specified name.

The search path is a string consisting of directory names separated by colons. `gawk` gets its search path from the `AWKPATH` environment variable. If that variable does not exist, `gawk` uses a default path, which is '`.:/usr/local/share/awk`'.[2] (Programs written for use by system administrators should use an `AWKPATH` variable that does not include the current directory, '`.`'.)

The search path feature is particularly useful for building up libraries of useful `awk` functions. The library files can be placed in a standard directory that is in the default path, and then specified on the command line with a short file name. Otherwise, the full file name would have to be typed for each file.

By using both the '`--source`' and '`-f`' options, your command line `awk` programs can use facilities in `awk` library files. See Chapter 15 [A Library of `awk` Functions], page 167.

Path searching is not done if `gawk` is in compatibility mode. This is true for both '`--traditional`' and '`--posix`'. See Section 14.1 [Command Line Options], page 159.

Note: if you want files in the current directory to be found, you must include the current directory in the path, either by including '`.`' explicitly in the path, or by writing a null entry in the path. (A null entry is indicated by starting or ending the path with a colon, or by placing two colons next to each other ('`::`').) If the current directory is not included in the path, then

[2] Your version of `gawk` may use a different directory; it will depend upon how `gawk` was built and installed. The actual directory will be the value of '`$(datadir)`' generated when `gawk` was configured. You probably don't need to worry about this though.

files cannot be found in the current directory. This path search mechanism is identical to the shell's.

Starting with version 3.0, if `AWKPATH` is not defined in the environment, `gawk` will place its default search path into `ENVIRON["AWKPATH"]`. This makes it easy to determine the actual search path `gawk` will use.

14.4 Obsolete Options and/or Features

This section describes features and/or command line options from previous releases of `gawk` that are either not available in the current version, or that are still supported but deprecated (meaning that they will *not* be in the next release).

For version 3.0.6 of `gawk`, there are no command line options or other deprecated features from the previous version of `gawk`. This section is thus essentially a place holder, in case some option becomes obsolete in a future version of `gawk`.

14.5 Undocumented Options and Features

> *Use the Source, Luke!*
> Obi-Wan

This section intentionally left blank.

14.6 Known Bugs in `gawk`

- The '`-F`' option for changing the value of `FS` (see Section 14.1 [Command Line Options], page 159) is not necessary given the command line variable assignment feature; it remains only for backwards compatibility.
- If your system actually has support for '`/dev/fd`' and the associated '`/dev/stdin`', '`/dev/stdout`', and '`/dev/stderr`' files, you may get different output from `gawk` than you would get on a system without those files. When `gawk` interprets these files internally, it synchronizes output to the standard output with output to '`/dev/stdout`', while on a system with those files, the output is actually to different open files (see Section 6.7 [Special File Names in `gawk`], page 69).
- Syntactically invalid single character programs tend to overflow the parse stack, generating a rather unhelpful message. Such programs are surprisingly difficult to diagnose in the completely general case, and the effort to do so really is not worth it.

15 A Library of awk Functions

This chapter presents a library of useful `awk` functions. The sample programs presented later (see Chapter 16 [Practical `awk` Programs], page 201) use these functions. The functions are presented here in a progression from simple to complex.

Section 16.2.7 [Extracting Programs from Texinfo Source Files], page 233, presents a program that you can use to extract the source code for these example library functions and programs from the Texinfo source for this book. (This has already been done as part of the `gawk` distribution.)

If you have written one or more useful, general purpose `awk` functions, and would like to contribute them for a subsequent edition of this book, please contact the author. See Section B.7 [Reporting Problems and Bugs], page 285, for information on doing this. Don't just send code, as you will be required to either place your code in the public domain, publish it under the GPL (see [GNU GENERAL PUBLIC LICENSE], page 303), or assign the copyright in it to the Free Software Foundation.

15.1 Simulating gawk-specific Features

The programs in this chapter and in Chapter 16 [Practical `awk` Programs], page 201, freely use features that are specific to `gawk`. This section briefly discusses how you can rewrite these programs for different implementations of `awk`.

Diagnostic error messages are sent to '`/dev/stderr`'. Use '`| "cat 1>&2"`' instead of '`> "/dev/stderr"`', if your system does not have a '`/dev/stderr`', or if you cannot use `gawk`.

A number of programs use `nextfile` (see Section 9.8 [The `nextfile` Statement], page 109), to skip any remaining input in the input file. Section 15.2 [Implementing `nextfile` as a Function], page 167, shows you how to write a function that will do the same thing.

Finally, some of the programs choose to ignore upper-case and lower-case distinctions in their input. They do this by assigning one to `IGNORECASE`. You can achieve the same effect by adding the following rule to the beginning of the program:

```
# ignore case
{ $0 = tolower($0) }
```

Also, verify that all regexp and string constants used in comparisons only use lower-case letters.

15.2 Implementing nextfile as a Function

The `nextfile` statement presented in Section 9.8 [The `nextfile` Statement], page 109, is a `gawk`-specific extension. It is not available in other

implementations of `awk`. This section shows two versions of a `nextfile` function that you can use to simulate `gawk`'s `nextfile` statement if you cannot use `gawk`.

Here is a first attempt at writing a `nextfile` function.

```
# nextfile --- skip remaining records in current file

# this should be read in before the "main" awk program

function nextfile()    { _abandon_ = FILENAME; next }

_abandon_ == FILENAME  { next }
```

This file should be included before the main program, because it supplies a rule that must be executed first. This rule compares the current data file's name (which is always in the `FILENAME` variable) to a private variable named `_abandon_`. If the file name matches, then the action part of the rule executes a `next` statement, to go on to the next record. (The use of '`_`' in the variable name is a convention. It is discussed more fully in Section 15.13 [Naming Library Function Global Variables], page 199.)

The use of the `next` statement effectively creates a loop that reads all the records from the current data file. Eventually, the end of the file is reached, and a new data file is opened, changing the value of `FILENAME`. Once this happens, the comparison of `_abandon_` to `FILENAME` fails, and execution continues with the first rule of the "real" program.

The `nextfile` function itself simply sets the value of `_abandon_` and then executes a `next` statement to start the loop going.[1]

This initial version has a subtle problem. What happens if the same data file is listed *twice* on the command line, one right after the other, or even with just a variable assignment between the two occurrences of the file name?

In such a case, this code will skip right through the file, a second time, even though it should stop when it gets to the end of the first occurrence. Here is a second version of `nextfile` that remedies this problem.

```
# nextfile --- skip remaining records in current file
# correctly handle successive occurrences of the same file
# Arnold Robbins, arnold@gnu.org, Public Domain
# May, 1993

# this should be read in before the "main" awk program

function nextfile()   { _abandon_ = FILENAME; next }
```

[1] Some implementations of `awk` do not allow you to execute `next` from within a function body. Some other work-around will be necessary if you use such a version.

```
_abandon_ == FILENAME {
      if (FNR == 1)
          _abandon_ = ""
      else
          next
}
```

The `nextfile` function has not changed. It sets `_abandon_` equal to the current file name and then executes a `next` satement. The `next` statement reads the next record and increments `FNR`, so `FNR` is guaranteed to have a value of at least two. However, if `nextfile` is called for the last record in the file, then `awk` will close the current data file and move on to the next one. Upon doing so, `FILENAME` will be set to the name of the new file, and `FNR` will be reset to one. If this next file is the same as the previous one, `_abandon_` will still be equal to `FILENAME`. However, `FNR` will be equal to one, telling us that this is a new occurrence of the file, and not the one we were reading when the `nextfile` function was executed. In that case, `_abandon_` is reset to the empty string, so that further executions of this rule will fail (until the next time that `nextfile` is called).

If `FNR` is not one, then we are still in the original data file, and the program executes a `next` statement to skip through it.

An important question to ask at this point is: "Given that the functionality of `nextfile` can be provided with a library file, why is it built into `gawk`?" This is an important question. Adding features for little reason leads to larger, slower programs that are harder to maintain.

The answer is that building `nextfile` into `gawk` provides significant gains in efficiency. If the `nextfile` function is executed at the beginning of a large data file, `awk` still has to scan the entire file, splitting it up into records, just to skip over it. The built-in `nextfile` can simply close the file immediately and proceed to the next one, saving a lot of time. This is particularly important in `awk`, since `awk` programs are generally I/O bound (i.e. they spend most of their time doing input and output, instead of performing computations).

15.3 Assertions

When writing large programs, it is often useful to be able to know that a condition or set of conditions is true. Before proceeding with a particular computation, you make a statement about what you believe to be the case. Such a statement is known as an "assertion." The C language provides an `<assert.h>` header file and corresponding `assert` macro that the programmer can use to make assertions. If an assertion fails, the `assert` macro arranges to print a diagnostic message describing the condition that should have been true but was not, and then it kills the program. In C, using `assert` looks this:

```
#include <assert.h>

int myfunc(int a, double b)
{
     assert(a <= 5 && b >= 17);
     ...
}
```

If the assertion failed, the program would print a message similar to this:

```
prog.c:5: assertion failed: a <= 5 && b >= 17
```

The ANSI C language makes it possible to turn the condition into a string for use in printing the diagnostic message. This is not possible in `awk`, so this `assert` function also requires a string version of the condition that is being tested.

```
# assert --- assert that a condition is true. Otherwise exit.
# Arnold Robbins, arnold@gnu.org, Public Domain
# May, 1993

function assert(condition, string)
{
    if (! condition) {
        printf("%s:%d: assertion failed: %s\n",
            FILENAME, FNR, string) > "/dev/stderr"
        _assert_exit = 1
        exit 1
    }
}

END {
    if (_assert_exit)
        exit 1
}
```

The `assert` function tests the `condition` parameter. If it is false, it prints a message to standard error, using the `string` parameter to describe the failed condition. It then sets the variable `_assert_exit` to one, and executes the `exit` statement. The `exit` statement jumps to the `END` rule. If the `END` rules finds `_assert_exit` to be true, then it exits immediately.

The purpose of the `END` rule with its test is to keep any other `END` rules from running. When an assertion fails, the program should exit immediately. If no assertions fail, then `_assert_exit` will still be false when the `END` rule is run normally, and the rest of the program's `END` rules will execute. For all of this to work correctly, '`assert.awk`' must be the first source file read by `awk`.

You would use this function in your programs this way:

```
function myfunc(a, b)
{
     assert(a <= 5 && b >= 17, "a <= 5 && b >= 17")
     ...
}
```

If the assertion failed, you would see a message like this:

```
mydata:1357: assertion failed: a <= 5 && b >= 17
```

There is a problem with this version of `assert`, that it may not be possible to work around with standard `awk`. An `END` rule is automatically added to the program calling `assert`. Normally, if a program consists of just a `BEGIN` rule, the input files and/or standard input are not read. However, now that the program has an `END` rule, `awk` will attempt to read the input data files, or standard input (see Section 8.1.5.1 [Startup and Cleanup Actions], page 98), most likely causing the program to hang, waiting for input.

15.4 Rounding Numbers

The way `printf` and `sprintf` (see Section 6.5 [Using `printf` Statements for Fancier Printing], page 62) do rounding will often depend upon the system's C `sprintf` subroutine. On many machines, `sprintf` rounding is "unbiased," which means it doesn't always round a trailing '`.5`' up, contrary to naive expectations. In unbiased rounding, '`.5`' rounds to even, rather than always up, so 1.5 rounds to 2 but 4.5 rounds to 4. The result is that if you are using a format that does rounding (e.g., `"%.0f"`) you should check what your system does. The following function does traditional rounding; it might be useful if your awk's `printf` does unbiased rounding.

```
# round --- do normal rounding
#
# Arnold Robbins, arnold@gnu.org, August, 1996
# Public Domain

function round(x,   ival, aval, fraction)
{
   ival = int(x)    # integer part, int() truncates

   # see if fractional part
   if (ival == x)   # no fraction
      return x

   if (x < 0) {
      aval = -x     # absolute value
      ival = int(aval)
      fraction = aval - ival
```

```
        if (fraction >= .5)
           return int(x) - 1   # -2.5 --> -3
        else
           return int(x)       # -2.3 --> -2
     } else {
        fraction = x - ival
        if (fraction >= .5)
           return ival + 1
        else
           return ival
     }
}

# test harness
{ print $0, round($0) }
```

15.5 Translating Between Characters and Numbers

One commercial implementation of `awk` supplies a built-in function, `ord`, which takes a character and returns the numeric value for that character in the machine's character set. If the string passed to `ord` has more than one character, only the first one is used.

The inverse of this function is `chr` (from the function of the same name in Pascal), which takes a number and returns the corresponding character.

Both functions can be written very nicely in `awk`; there is no real reason to build them into the `awk` interpreter.

```
# ord.awk --- do ord and chr
#
# Global identifiers:
#    _ord_:        numerical values indexed by characters
#    _ord_init:    function to initialize _ord_
#
# Arnold Robbins
# arnold@gnu.org
# Public Domain
# 16 January, 1992
# 20 July, 1992, revised

BEGIN    { _ord_init() }

function _ord_init(    low, high, i, t)
{
    low = sprintf("%c", 7) # BEL is ascii 7
```

```
    if (low == "\a") {    # regular ascii
        low = 0
        high = 127
    } else if (sprintf("%c", 128 + 7) == "\a") {
        # ascii, mark parity
        low = 128
        high = 255
    } else {        # ebcdic(!)
        low = 0
        high = 255
    }

    for (i = low; i <= high; i++) {
        t = sprintf("%c", i)
        _ord_[t] = i
    }
}
```

Some explanation of the numbers used by `chr` is worthwhile. The most prominent character set in use today is ASCII. Although an eight-bit byte can hold 256 distinct values (from zero to 255), ASCII only defines characters that use the values from zero to 127.[1] At least one computer manufacturer that we know of uses ASCII, but with mark parity, meaning that the leftmost bit in the byte is always one. What this means is that on those systems, characters have numeric values from 128 to 255. Finally, large mainframe systems use the EBCDIC character set, which uses all 256 values. While there are other character sets in use on some older systems, they are not really worth worrying about.

```
function ord(str,    c)
{
    # only first character is of interest
    c = substr(str, 1, 1)
    return _ord_[c]
}

function chr(c)
{
    # force c to be numeric by adding 0
    return sprintf("%c", c + 0)
}
```

[1] ASCII has been extended in many countries to use the values from 128 to 255 for country-specific characters. If your system uses these extensions, you can simplify `_ord_init` to simply loop from zero to 255.

```
#### test code ####
# BEGIN    \
# {
#    for (;;) {
#        printf("enter a character: ")
#        if (getline var <= 0)
#            break
#        printf("ord(%s) = %d\n", var, ord(var))
#    }
# }
```

An obvious improvement to these functions would be to move the code for the `_ord_init` function into the body of the `BEGIN` rule. It was written this way initially for ease of development.

There is a "test program" in a `BEGIN` rule, for testing the function. It is commented out for production use.

15.6 Merging an Array Into a String

When doing string processing, it is often useful to be able to join all the strings in an array into one long string. The following function, `join`, accomplishes this task. It is used later in several of the application programs (see Chapter 16 [Practical `awk` Programs], page 201).

Good function design is important; this function needs to be general, but it should also have a reasonable default behavior. It is called with an array and the beginning and ending indices of the elements in the array to be merged. This assumes that the array indices are numeric—a reasonable assumption since the array was likely created with `split` (see Section 12.3 [Built-in Functions for String Manipulation], page 135).

```
# join.awk --- join an array into a string
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

function join(array, start, end, sep,    result, i)
{
    if (sep == "")
       sep = " "
    else if (sep == SUBSEP) # magic value
       sep = ""
    result = array[start]
    for (i = start + 1; i <= end; i++)
        result = result sep array[i]
    return result
}
```

An optional additional argument is the separator to use when joining the strings back together. If the caller supplies a non-empty value, `join` uses it. If it is not supplied, it will have a null value. In this case, `join` uses a single blank as a default separator for the strings. If the value is equal to `SUBSEP`, then `join` joins the strings with no separator between them. `SUBSEP` serves as a "magic" value to indicate that there should be no separation between the component strings.

It would be nice if `awk` had an assignment operator for concatenation. The lack of an explicit operator for concatenation makes string operations more difficult than they really need to be.

15.7 Turning Dates Into Timestamps

The `systime` function built in to `gawk` returns the current time of day as a timestamp in "seconds since the Epoch." This timestamp can be converted into a printable date of almost infinitely variable format using the built-in `strftime` function. (For more information on `systime` and `strftime`, see Section 12.5 [Functions for Dealing with Time Stamps], page 146.)

An interesting but difficult problem is to convert a readable representation of a date back into a timestamp. The ANSI C library provides a `mktime` function that does the basic job, converting a canonical representation of a date into a timestamp.

It would appear at first glance that `gawk` would have to supply a `mktime` built-in function that was simply a "hook" to the C language version. In fact though, `mktime` can be implemented entirely in `awk`.[2]

Here is a version of `mktime` for `awk`. It takes a simple representation of the date and time, and converts it into a timestamp.

The code is presented here intermixed with explanatory prose. In Section 16.2.7 [Extracting Programs from Texinfo Source Files], page 233, you will see how the Texinfo source file for this book can be processed to extract the code into a single source file.

The program begins with a descriptive comment and a `BEGIN` rule that initializes a table `_tm_months`. This table is a two-dimensional array that has the lengths of the months. The first index is zero for regular years, and one for leap years. The values are the same for all the months in both kinds of years, except for February; thus the use of multiple assignment.

```
# mktime.awk --- convert a canonical date representation
#                into a timestamp
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993
```

[2] July, 2000: Actually, I was mistaken when I wrote this. The version presented here doesn't always work correctly, and the next major version of `gawk` will provide `mktime` as a built-in function.

```
BEGIN    \
{
    # Initialize table of month lengths
    _tm_months[0,1] = _tm_months[1,1] = 31
    _tm_months[0,2] = 28; _tm_months[1,2] = 29
    _tm_months[0,3] = _tm_months[1,3] = 31
    _tm_months[0,4] = _tm_months[1,4] = 30
    _tm_months[0,5] = _tm_months[1,5] = 31
    _tm_months[0,6] = _tm_months[1,6] = 30
    _tm_months[0,7] = _tm_months[1,7] = 31
    _tm_months[0,8] = _tm_months[1,8] = 31
    _tm_months[0,9] = _tm_months[1,9] = 30
    _tm_months[0,10] = _tm_months[1,10] = 31
    _tm_months[0,11] = _tm_months[1,11] = 30
    _tm_months[0,12] = _tm_months[1,12] = 31
}
```

The benefit of merging multiple `BEGIN` rules (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98) is particularly clear when writing library files. Functions in library files can cleanly initialize their own private data and also provide clean-up actions in private `END` rules.

The next function is a simple one that computes whether a given year is or is not a leap year. If a year is evenly divisible by four, but not evenly divisible by 100, or if it is evenly divisible by 400, then it is a leap year. Thus, 1904 was a leap year, 1900 was not, but 2000 will be.

```
# decide if a year is a leap year
function _tm_isleap(year,    ret)
{
    ret = (year % 4 == 0 && year % 100 != 0) ||
            (year % 400 == 0)

    return ret
}
```

This function is only used a few times in this file, and its computation could have been written *in-line* (at the point where it's used). Making it a separate function made the original development easier, and also avoids the possibility of typing errors when duplicating the code in multiple places.

The next function is more interesting. It does most of the work of generating a timestamp, which is converting a date and time into some number of seconds since the Epoch. The caller passes an array (rather imaginatively named `a`) containing six values: the year including century, the month as a number between one and 12, the day of the month, the hour as a number between zero and 23, the minute in the hour, and the seconds within the minute.

The function uses several local variables to precompute the number of seconds in an hour, seconds in a day, and seconds in a year. Often, similar C code simply writes out the expression in-line, expecting the compiler to do *constant folding*. E.g., most C compilers would turn '`60 * 60`' into '`3600`' at compile time, instead of recomputing it every time at run time. Precomputing these values makes the function more efficient.

```
# convert a date into seconds
function _tm_addup(a,    total, yearsecs, daysecs,
                         hoursecs, i, j)
{
    hoursecs = 60 * 60
    daysecs = 24 * hoursecs
    yearsecs = 365 * daysecs

    total = (a[1] - 1970) * yearsecs

    # extra day for leap years
    for (i = 1970; i < a[1]; i++)
        if (_tm_isleap(i))
            total += daysecs

    j = _tm_isleap(a[1])
    for (i = 1; i < a[2]; i++)
        total += _tm_months[j, i] * daysecs

    total += (a[3] - 1) * daysecs
    total += a[4] * hoursecs
    total += a[5] * 60
    total += a[6]

    return total
}
```

The function starts with a first approximation of all the seconds between Midnight, January 1, 1970,[3] and the beginning of the current year. It then goes through all those years, and for every leap year, adds an additional day's worth of seconds.

The variable `j` holds either one or zero, if the current year is or is not a leap year. For every month in the current year prior to the current month, it adds the number of seconds in the month, using the appropriate entry in the `_tm_months` array.

Finally, it adds in the seconds for the number of days prior to the current day, and the number of hours, minutes, and seconds in the current day.

[3] This is the Epoch on POSIX systems. It may be different on other systems.

The result is a count of seconds since January 1, 1970. This value is not yet what is needed though. The reason why is described shortly.

The main `mktime` function takes a single character string argument. This string is a representation of a date and time in a "canonical" (fixed) form. This string should be "*year month day hour minute second*".

```
# mktime --- convert a date into seconds,
#            compensate for time zone

function mktime(str,    res1, res2, a, b, i, j, t, diff)
{
    i = split(str, a, " ")    # don't rely on FS

    if (i != 6)
        return -1

    # force numeric
    for (j in a)
        a[j] += 0

    # validate
    if (a[1] < 1970 ||
        a[2] < 1 || a[2] > 12 ||
        a[3] < 1 || a[3] > 31 ||
        a[4] < 0 || a[4] > 23 ||
        a[5] < 0 || a[5] > 59 ||
        a[6] < 0 || a[6] > 60 )
            return -1

    res1 = _tm_addup(a)
    t = strftime("%Y %m %d %H %M %S", res1)

    if (_tm_debug)
        printf("(%s) -> (%s)\n", str, t) > "/dev/stderr"

    split(t, b, " ")
    res2 = _tm_addup(b)

    diff = res1 - res2

    if (_tm_debug)
        printf("diff = %d seconds\n", diff) > "/dev/stderr"

    res1 += diff
```

```
    return res1
}
```

The function first splits the string into an array, using spaces and tabs as separators. If there are not six elements in the array, it returns an error, signaled as the value -1. Next, it forces each element of the array to be numeric, by adding zero to it. The following '`if`' statement then makes sure that each element is within an allowable range. (This checking could be extended further, e.g., to make sure that the day of the month is within the correct range for the particular month supplied.) All of this is essentially preliminary set-up and error checking.

Recall that `_tm_addup` generated a value in seconds since Midnight, January 1, 1970. This value is not directly usable as the result we want, *since the calculation does not account for the local timezone.* In other words, the value represents the count in seconds since the Epoch, but only for UTC (Universal Coordinated Time). If the local timezone is east or west of UTC, then some number of hours should be either added to, or subtracted from the resulting timestamp.

For example, 6:23 p.m. in Atlanta, Georgia (USA), is normally five hours west of (behind) UTC. It is only four hours behind UTC if daylight savings time is in effect. If you are calling `mktime` in Atlanta, with the argument `"1993 5 23 18 23 12"`, the result from `_tm_addup` will be for 6:23 p.m. UTC, which is only 2:23 p.m. in Atlanta. It is necessary to add another four hours worth of seconds to the result.

How can `mktime` determine how far away it is from UTC? This is surprisingly easy. The returned timestamp represents the time passed to `mktime` *as UTC.* This timestamp can be fed back to `strftime`, which will format it as a *local* time; i.e. as if it already had the UTC difference added in to it. This is done by giving `"%Y %m %d %H %M %S"` to `strftime` as the format argument. It returns the computed timestamp in the original string format. The result represents a time that accounts for the UTC difference. When the new time is converted back to a timestamp, the difference between the two timestamps is the difference (in seconds) between the local timezone and UTC. This difference is then added back to the original result. An example demonstrating this is presented below.

Finally, there is a "main" program for testing the function.

```
BEGIN  {
    if (_tm_test) {
        printf "Enter date as yyyy mm dd hh mm ss: "
        getline _tm_test_date
        t = mktime(_tm_test_date)
        r = strftime("%Y %m %d %H %M %S", t)
        printf "Got back (%s)\n", r
    }
}
```

The entire program uses two variables that can be set on the command line to control debugging output and to enable the test in the final `BEGIN` rule. Here is the result of a test run. (Note that debugging output is to standard error, and test output is to standard output.)

```
$ gawk -f mktime.awk -v _tm_test=1 -v _tm_debug=1
⊣ Enter date as yyyy mm dd hh mm ss: 1993 5 23 15 35 10
error  (1993 5 23 15 35 10) -> (1993 05 23 11 35 10)
error  diff = 14400 seconds
⊣ Got back (1993 05 23 15 35 10)
```

The time entered was 3:35 p.m. (15:35 on a 24-hour clock), on May 23, 1993. The first line of debugging output shows the resulting time as UTC—four hours ahead of the local time zone. The second line shows that the difference is 14400 seconds, which is four hours. (The difference is only four hours, since daylight savings time is in effect during May.) The final line of test output shows that the timezone compensation algorithm works; the returned time is the same as the entered time.

This program does not solve the general problem of turning an arbitrary date representation into a timestamp. That problem is very involved. However, the `mktime` function provides a foundation upon which to build. Other software can convert month names into numeric months, and AM/PM times into 24-hour clocks, to generate the "canonical" format that `mktime` requires.

15.8 Managing the Time of Day

The `systime` and `strftime` functions described in Section 12.5 [Functions for Dealing with Time Stamps], page 146, provide the minimum functionality necessary for dealing with the time of day in human readable form. While `strftime` is extensive, the control formats are not necessarily easy to remember or intuitively obvious when reading a program.

The following function, `gettimeofday`, populates a user-supplied array with pre-formatted time information. It returns a string with the current time formatted in the same way as the `date` utility.

```
# gettimeofday --- get the time of day in a usable format
# Arnold Robbins, arnold@gnu.org, Public Domain, May 1993
#
# Returns a string in the format of output of date(1)
# Populates the array argument time with individual values:
#    time["second"]       -- seconds (0 - 59)
#    time["minute"]       -- minutes (0 - 59)
#    time["hour"]         -- hours (0 - 23)
#    time["althour"]      -- hours (0 - 12)
#    time["monthday"]     -- day of month (1 - 31)
#    time["month"]        -- month of year (1 - 12)
#    time["monthname"]    -- name of the month
```

```
#    time["shortmonth"]   -- short name of the month
#    time["year"]         -- year within century (0 - 99)
#    time["fullyear"]     -- year with century (19xx or 20xx)
#    time["weekday"]      -- day of week (Sunday = 0)
#    time["altweekday"]   -- day of week (Monday = 0)
#    time["weeknum"]      -- week number, Sunday first day
#    time["altweeknum"]   -- week number, Monday first day
#    time["dayname"]      -- name of weekday
#    time["shortdayname"] -- short name of weekday
#    time["yearday"]      -- day of year (0 - 365)
#    time["timezone"]     -- abbreviation of timezone name
#    time["ampm"]         -- AM or PM designation

function gettimeofday(time,    ret, now, i)
{
    # get time once, avoids unnecessary system calls
    now = systime()

    # return date(1)-style output
    ret = strftime("%a %b %d %H:%M:%S %Z %Y", now)

    # clear out target array
    for (i in time)
        delete time[i]

    # fill in values, force numeric values to be
    # numeric by adding 0
    time["second"]       = strftime("%S", now) + 0
    time["minute"]       = strftime("%M", now) + 0
    time["hour"]         = strftime("%H", now) + 0
    time["althour"]      = strftime("%I", now) + 0
    time["monthday"]     = strftime("%d", now) + 0
    time["month"]        = strftime("%m", now) + 0
    time["monthname"]    = strftime("%B", now)
    time["shortmonth"]   = strftime("%b", now)
    time["year"]         = strftime("%y", now) + 0
    time["fullyear"]     = strftime("%Y", now) + 0
    time["weekday"]      = strftime("%w", now) + 0
    time["altweekday"]   = strftime("%u", now) + 0
    time["dayname"]      = strftime("%A", now)
    time["shortdayname"] = strftime("%a", now)
    time["yearday"]      = strftime("%j", now) + 0
    time["timezone"]     = strftime("%Z", now)
    time["ampm"]         = strftime("%p", now)
```

```
    time["weeknum"]      = strftime("%U", now) + 0
    time["altweeknum"]   = strftime("%W", now) + 0

    return ret
}
```

The string indices are easier to use and read than the various formats required by `strftime`. The `alarm` program presented in Section 16.2.2 [An Alarm Clock Program], page 223, uses this function.

The `gettimeofday` function is presented above as it was written. A more general design for this function would have allowed the user to supply an optional timestamp value that would have been used instead of the current time.

15.9 Noting Data File Boundaries

The `BEGIN` and `END` rules are each executed exactly once, at the beginning and end respectively of your `awk` program (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98). We (the `gawk` authors) once had a user who mistakenly thought that the `BEGIN` rule was executed at the beginning of each data file and the `END` rule was executed at the end of each data file. When informed that this was not the case, the user requested that we add new special patterns to `gawk`, named `BEGIN_FILE` and `END_FILE`, that would have the desired behavior. He even supplied us the code to do so.

However, after a little thought, I came up with the following library program. It arranges to call two user-supplied functions, `beginfile` and `endfile`, at the beginning and end of each data file. Besides solving the problem in only nine(!) lines of code, it does so *portably*; this will work with any implementation of `awk`.

```
# transfile.awk
#
# Give the user a hook for filename transitions
#
# The user must supply functions beginfile() and endfile()
# that each take the name of the file being started or
# finished, respectively.
#
# Arnold Robbins, arnold@gnu.org, January 1992
# Public Domain

FILENAME != _oldfilename \
{
    if (_oldfilename != "")
        endfile(_oldfilename)
    _oldfilename = FILENAME
```

```
    beginfile(FILENAME)
}

END   { endfile(FILENAME) }
```

This file must be loaded before the user's "main" program, so that the rule it supplies will be executed first.

This rule relies on `awk`'s `FILENAME` variable that automatically changes for each new data file. The current file name is saved in a private variable, `_oldfilename`. If `FILENAME` does not equal `_oldfilename`, then a new data file is being processed, and it is necessary to call `endfile` for the old file. Since `endfile` should only be called if a file has been processed, the program first checks to make sure that `_oldfilename` is not the null string. The program then assigns the current file name to `_oldfilename`, and calls `beginfile` for the file. Since, like all `awk` variables, `_oldfilename` will be initialized to the null string, this rule executes correctly even for the first data file.

The program also supplies an `END` rule, to do the final processing for the last file. Since this `END` rule comes before any `END` rules supplied in the "main" program, `endfile` will be called first. Once again the value of multiple `BEGIN` and `END` rules should be clear.

This version has same problem as the first version of `nextfile` (see Section 15.2 [Implementing `nextfile` as a Function], page 167). If the same data file occurs twice in a row on command line, then `endfile` and `beginfile` will not be executed at the end of the first pass and at the beginning of the second pass. This version solves the problem.

```
# ftrans.awk --- handle data file transitions
#
# user supplies beginfile() and endfile() functions
#
# Arnold Robbins, arnold@gnu.org, November 1992
# Public Domain

FNR == 1 {
    if (_filename_ != "")
        endfile(_filename_)
    _filename_ = FILENAME
    beginfile(FILENAME)
}

END  { endfile(_filename_) }
```

In Section 16.1.7 [Counting Things], page 220, you will see how this library function can be used, and how it simplifies writing the main program.

15.10 Processing Command Line Options

Most utilities on POSIX compatible systems take options or "switches" on the command line that can be used to change the way a program behaves. `awk` is an example of such a program (see Section 14.1 [Command Line Options], page 159). Often, options take *arguments*, data that the program needs to correctly obey the command line option. For example, `awk`'s '`-F`' option requires a string to use as the field separator. The first occurrence on the command line of either '`--`' or a string that does not begin with '`-`' ends the options.

Most Unix systems provide a C function named `getopt` for processing command line arguments. The programmer provides a string describing the one letter options. If an option requires an argument, it is followed in the string with a colon. `getopt` is also passed the count and values of the command line arguments, and is called in a loop. `getopt` processes the command line arguments for option letters. Each time around the loop, it returns a single character representing the next option letter that it found, or '`?`' if it found an invalid option. When it returns -1, there are no options left on the command line.

When using `getopt`, options that do not take arguments can be grouped together. Furthermore, options that take arguments require that the argument be present. The argument can immediately follow the option letter, or it can be a separate command line argument.

Given a hypothetical program that takes three command line options, '`-a`', '`-b`', and '`-c`', and '`-b`' requires an argument, all of the following are valid ways of invoking the program:

```
prog -a -b foo -c data1 data2 data3
prog -ac -bfoo -- data1 data2 data3
prog -acbfoo data1 data2 data3
```

Notice that when the argument is grouped with its option, the rest of the command line argument is considered to be the option's argument. In the above example, '`-acbfoo`' indicates that all of the '`-a`', '`-b`', and '`-c`' options were supplied, and that '`foo`' is the argument to the '`-b`' option.

`getopt` provides four external variables that the programmer can use.

`optind` The index in the argument value array (`argv`) where the first non-option command line argument can be found.

`optarg` The string value of the argument to an option.

`opterr` Usually `getopt` prints an error message when it finds an invalid option. Setting `opterr` to zero disables this feature. (An application might wish to print its own error message.)

`optopt` The letter representing the command line option. While not usually documented, most versions supply this variable.

The following C fragment shows how `getopt` might process command line arguments for `awk`.

```
int
main(int argc, char *argv[])
{
    ...
    /* print our own message */
    opterr = 0;
    while ((c = getopt(argc, argv, "v:f:F:W:")) != -1) {
        switch (c) {
        case 'f':    /* file */
            ...
            break;
        case 'F':    /* field separator */
            ...
            break;
        case 'v':    /* variable assignment */
            ...
            break;
        case 'W':    /* extension */
            ...
            break;
        case '?':
        default:
            usage();
            break;
        }
    }
    ...
}
```

As a side point, `gawk` actually uses the GNU `getopt_long` function to process both normal and GNU-style long options (see Section 14.1 [Command Line Options], page 159).

The abstraction provided by `getopt` is very useful, and would be quite handy in `awk` programs as well. Here is an `awk` version of `getopt`. This function highlights one of the greatest weaknesses in `awk`, which is that it is very poor at manipulating single characters. Repeated calls to `substr` are necessary for accessing individual characters (see Section 12.3 [Built-in Functions for String Manipulation], page 135).

The discussion walks through the code a bit at a time.

```
# getopt --- do C library getopt(3) function in awk
#
# arnold@gnu.org
```

```
# Public domain
#
# Initial version: March, 1991
# Revised: May, 1993

# External variables:
#    Optind -- index of ARGV for first non-option argument
#    Optarg -- string value of argument to current option
#    Opterr -- if non-zero, print our own diagnostic
#    Optopt -- current option letter

# Returns
#    -1     at end of options
#    ?      for unrecognized option
#    <c>    a character representing the current option

# Private Data
#    _opti  index in multi-flag option, e.g., -abc
```

The function starts out with some documentation: who wrote the code, and when it was revised, followed by a list of the global variables it uses, what the return values are and what they mean, and any global variables that are "private" to this library function. Such documentation is essential for any program, and particularly for library functions.

```
function getopt(argc, argv, options,    optl, thisopt, i)
{
    optl = length(options)
    if (optl == 0)        # no options given
        return -1

    if (argv[Optind] == "--") {  # all done
        Optind++
        _opti = 0
        return -1
    } else if (argv[Optind] !~ /^-[^: \t\n\f\r\v\b]/) {
        _opti = 0
        return -1
    }
```

The function first checks that it was indeed called with a string of options (the `options` parameter). If `options` has a zero length, `getopt` immediately returns -1.

The next thing to check for is the end of the options. A '`--`' ends the command line options, as does any command line argument that does not begin with a '`-`'. `Optind` is used to step through the array of command line

arguments; it retains its value across calls to `getopt`, since it is a global variable.

The regexp used, `/^-[^: \t\n\f\r\v\b]/`, is perhaps a bit of overkill; it checks for a '`-`' followed by anything that is not whitespace and not a colon. If the current command line argument does not match this pattern, it is not an option, and it ends option processing.

```
if (_opti == 0)
    _opti = 2
thisopt = substr(argv[Optind], _opti, 1)
Optopt = thisopt
i = index(options, thisopt)
if (i == 0) {
    if (Opterr)
        printf("%c -- invalid option\n",
                                thisopt) > "/dev/stderr"
    if (_opti >= length(argv[Optind])) {
        Optind++
        _opti = 0
    } else
        _opti++
    return "?"
}
```

The `_opti` variable tracks the position in the current command line argument (`argv[Optind]`). In the case that multiple options were grouped together with one '`-`' (e.g., '`-abx`'), it is necessary to return them to the user one at a time.

If `_opti` is equal to zero, it is set to two, the index in the string of the next character to look at (we skip the '`-`', which is at position one). The variable `thisopt` holds the character, obtained with `substr`. It is saved in `Optopt` for the main program to use.

If `thisopt` is not in the `options` string, then it is an invalid option. If `Opterr` is non-zero, `getopt` prints an error message on the standard error that is similar to the message from the C version of `getopt`.

Since the option is invalid, it is necessary to skip it and move on to the next option character. If `_opti` is greater than or equal to the length of the current command line argument, then it is necessary to move on to the next one, so `Optind` is incremented and `_opti` is reset to zero. Otherwise, `Optind` is left alone and `_opti` is merely incremented.

In any case, since the option was invalid, `getopt` returns '`?`'. The main program can examine `Optopt` if it needs to know what the invalid option letter actually was.

```
        if (substr(options, i + 1, 1) == ":") {
            # get option argument
            if (length(substr(argv[Optind], _opti + 1)) > 0)
                Optarg = substr(argv[Optind], _opti + 1)
            else
                Optarg = argv[++Optind]
            _opti = 0
        } else
            Optarg = ""
```

If the option requires an argument, the option letter is followed by a colon in the `options` string. If there are remaining characters in the current command line argument (`argv[Optind]`), then the rest of that string is assigned to `Optarg`. Otherwise, the next command line argument is used ('`-xFOO`' vs. '`-x FOO`'). In either case, `_opti` is reset to zero, since there are no more characters left to examine in the current command line argument.

```
        if (_opti == 0 || _opti >= length(argv[Optind])) {
            Optind++
            _opti = 0
        } else
            _opti++
        return thisopt
    }
```

Finally, if `_opti` is either zero or greater than the length of the current command line argument, it means this element in `argv` is through being processed, so `Optind` is incremented to point to the next element in `argv`. If neither condition is true, then only `_opti` is incremented, so that the next option letter can be processed on the next call to `getopt`.

```
    BEGIN {
        Opterr = 1    # default is to diagnose
        Optind = 1    # skip ARGV[0]

        # test program
        if (_getopt_test) {
            while ((_go_c = getopt(ARGC, ARGV, "ab:cd")) != -1)
                printf("c = <%c>, optarg = <%s>\n",
                                           _go_c, Optarg)
            printf("non-option arguments:\n")
            for (; Optind < ARGC; Optind++)
                printf("\tARGV[%d] = <%s>\n",
                                        Optind, ARGV[Optind])
        }
    }
```

The `BEGIN` rule initializes both `Opterr` and `Optind` to one. `Opterr` is set to one, since the default behavior is for `getopt` to print a diagnostic message

upon seeing an invalid option. `Optind` is set to one, since there's no reason to look at the program name, which is in `ARGV[0]`.

The rest of the `BEGIN` rule is a simple test program. Here is the result of two sample runs of the test program.

```
$ awk -f getopt.awk -v _getopt_test=1 -- -a -cbARG bax -x
⊣ c = <a>, optarg = <>
⊣ c = <c>, optarg = <>
⊣ c = <b>, optarg = <ARG>
⊣ non-option arguments:
⊣         ARGV[3] = <bax>
⊣         ARGV[4] = <-x>

$ awk -f getopt.awk -v _getopt_test=1 -- -a -x -- xyz abc
⊣ c = <a>, optarg = <>
error  x -- invalid option
⊣ c = <?>, optarg = <>
⊣ non-option arguments:
⊣         ARGV[4] = <xyz>
⊣         ARGV[5] = <abc>
```

The first '`--`' terminates the arguments to `awk`, so that it does not try to interpret the '`-a`' etc. as its own options.

Several of the sample programs presented in Chapter 16 [Practical `awk` Programs], page 201, use `getopt` to process their arguments.

15.11 Reading the User Database

The '`/dev/user`' special file (see Section 6.7 [Special File Names in `gawk`], page 69) provides access to the current user's real and effective user and group id numbers, and if available, the user's supplementary group set. However, since these are numbers, they do not provide very useful information to the average user. There needs to be some way to find the user information associated with the user and group numbers. This section presents a suite of functions for retrieving information from the user database. See Section 15.12 [Reading the Group Database], page 194, for a similar suite that retrieves information from the group database.

The POSIX standard does not define the file where user information is kept. Instead, it provides the `<pwd.h>` header file and several C language subroutines for obtaining user information. The primary function is `getpwent`, for "get password entry." The "password" comes from the original user database file, '`/etc/passwd`', which kept user information, along with the encrypted passwords (hence the name).

While an `awk` program could simply read '`/etc/passwd`' directly (the format is well known), because of the way password files are handled on networked systems, this file may not contain complete information about the system's set of users.

To be sure of being able to produce a readable, complete version of the user database, it is necessary to write a small C program that calls `getpwent`. `getpwent` is defined to return a pointer to a `struct passwd`. Each time it is called, it returns the next entry in the database. When there are no more entries, it returns `NULL`, the null pointer. When this happens, the C program should call `endpwent` to close the database. Here is `pwcat`, a C program that "cats" the password database.

```
/*
 * pwcat.c
 *
 * Generate a printable version of the password database
 *
 * Arnold Robbins
 * arnold@gnu.org
 * May 1993
 * Public Domain
 */

#include <stdio.h>
#include <pwd.h>

int
main(argc, argv)
int argc;
char **argv;
{
    struct passwd *p;

    while ((p = getpwent()) != NULL)
        printf("%s:%s:%d:%d:%s:%s:%s\n",
            p->pw_name, p->pw_passwd, p->pw_uid,
            p->pw_gid, p->pw_gecos, p->pw_dir, p->pw_shell);

    endpwent();
    exit(0);
}
```

If you don't understand C, don't worry about it. The output from `pwcat` is the user database, in the traditional '`/etc/passwd`' format of colon-separated fields. The fields are:

Login name
: The user's login name.

Encrypted password
: The user's encrypted password. This may not be available on some systems.

User-ID
: The user's numeric user-id number.

Group-ID
: The user's numeric group-id number.

Full name
: The user's full name, and perhaps other information associated with the user.

Home directory
: The user's login, or "home" directory (familiar to shell programmers as `$HOME`).

Login shell
: The program that will be run when the user logs in. This is usually a shell, such as Bash (the Gnu Bourne-Again shell).

Here are a few lines representative of `pwcat`'s output.

```
$ pwcat
⊣ root:3Ov02d5VaUPB6:0:1:Operator:/:/bin/sh
⊣ nobody:*:65534:65534::/:
⊣ daemon:*:1:1::/:
⊣ sys:*:2:2::/:/bin/csh
⊣ bin:*:3:3::/bin:
⊣ arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh
⊣ miriam:yxaay:112:10:Miriam Robbins:/home/miriam:/bin/sh
⊣ andy:abcca2:113:10:Andy Jacobs:/home/andy:/bin/sh
...
```

With that introduction, here is a group of functions for getting user information. There are several functions here, corresponding to the C functions of the same name.

```
# passwd.awk --- access password file information
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

BEGIN {
    # tailor this to suit your system
    _pw_awklib = "/usr/local/libexec/awk/"
}
```

```
function _pw_init(    oldfs, oldrs, olddol0, pwcat)
{
    if (_pw_inited)
        return
    oldfs = FS
    oldrs = RS
    olddol0 = $0
    FS = ":"
    RS = "\n"
    pwcat = _pw_awklib "pwcat"
    while ((pwcat | getline) > 0) {
        _pw_byname[$1] = $0
        _pw_byuid[$3] = $0
        _pw_bycount[++_pw_total] = $0
    }
    close(pwcat)
    _pw_count = 0
    _pw_inited = 1
    FS = oldfs
    RS = oldrs
    $0 = olddol0
}
```

The `BEGIN` rule sets a private variable to the directory where `pwcat` is stored. Since it is used to help out an `awk` library routine, we have chosen to put it in '`/usr/local/libexec/awk`'. You might want it to be in a different directory on your system.

The function `_pw_init` keeps three copies of the user information in three associative arrays. The arrays are indexed by user name (`_pw_byname`), by user-id number (`_pw_byuid`), and by order of occurrence (`_pw_bycount`).

The variable `_pw_inited` is used for efficiency; `_pw_init` only needs to be called once.

Since this function uses `getline` to read information from `pwcat`, it first saves the values of `FS`, `RS`, and `$0`. Doing so is necessary, since these functions could be called from anywhere within a user's program, and the user may have his or her own values for `FS` and `RS`.

The main part of the function uses a loop to read database lines, split the line into fields, and then store the line into each array as necessary. When the loop is done, `_pw_init` cleans up by closing the pipeline, setting `_pw_inited` to one, and restoring `FS`, `RS`, and `$0`. The use of `_pw_count` will be explained below.

```
function getpwnam(name)
{
    _pw_init()
    if (name in _pw_byname)
        return _pw_byname[name]
    return ""
}
```

The `getpwnam` function takes a user name as a string argument. If that user is in the database, it returns the appropriate line. Otherwise it returns the null string.

```
function getpwuid(uid)
{
    _pw_init()
    if (uid in _pw_byuid)
        return _pw_byuid[uid]
    return ""
}
```

Similarly, the `getpwuid` function takes a user-id number argument. If that user number is in the database, it returns the appropriate line. Otherwise it returns the null string.

```
function getpwent()
{
    _pw_init()
    if (_pw_count < _pw_total)
        return _pw_bycount[++_pw_count]
    return ""
}
```

The `getpwent` function simply steps through the database, one entry at a time. It uses `_pw_count` to track its current position in the `_pw_bycount` array.

```
function endpwent()
{
    _pw_count = 0
}
```

The `endpwent` function resets `_pw_count` to zero, so that subsequent calls to `getpwent` will start over again.

A conscious design decision in this suite is that each subroutine calls `_pw_init` to initialize the database arrays. The overhead of running a separate process to generate the user database, and the I/O to scan it, will only be incurred if the user's main program actually calls one of these functions. If this library file is loaded along with a user's program, but none of the routines are ever called, then there is no extra run-time overhead. (The alternative would be to move the body of `_pw_init` into a `BEGIN` rule, which

would always run `pwcat`. This simplifies the code but runs an extra process that may never be needed.)

In turn, calling `_pw_init` is not too expensive, since the `_pw_inited` variable keeps the program from reading the data more than once. If you are worried about squeezing every last cycle out of your `awk` program, the check of `_pw_inited` could be moved out of `_pw_init` and duplicated in all the other functions. In practice, this is not necessary, since most `awk` programs are I/O bound, and it would clutter up the code.

The `id` program in Section 16.1.3 [Printing Out User Information], page 210, uses these functions.

15.12 Reading the Group Database

Much of the discussion presented in Section 15.11 [Reading the User Database], page 189, applies to the group database as well. Although there has traditionally been a well known file, '`/etc/group`', in a well known format, the POSIX standard only provides a set of C library routines (`<grp.h>` and `getgrent`) for accessing the information. Even though this file may exist, it likely does not have complete information. Therefore, as with the user database, it is necessary to have a small C program that generates the group database as its output.

Here is `grcat`, a C program that "cats" the group database.

```
/*
 * grcat.c
 *
 * Generate a printable version of the group database
 *
 * Arnold Robbins, arnold@gnu.org
 * May 1993
 * Public Domain
 */

#include <stdio.h>
#include <grp.h>

int
main(argc, argv)
int argc;
char **argv;
{
    struct group *g;
    int i;
```

```
    while ((g = getgrent()) != NULL) {
        printf("%s:%s:%d:", g->gr_name, g->gr_passwd,
                                            g->gr_gid);
        for (i = 0; g->gr_mem[i] != NULL; i++) {
            printf("%s", g->gr_mem[i]);
            if (g->gr_mem[i+1] != NULL)
                putchar(',');
        }
        putchar('\n');
    }
    endgrent();
    exit(0);
}
```

Each line in the group database represent one group. The fields are separated with colons, and represent the following information.

Group Name
: The name of the group.

Group Password
: The encrypted group password. In practice, this field is never used. It is usually empty, or set to '`*`'.

Group ID Number
: The numeric group-id number. This number should be unique within the file.

Group Member List
: A comma-separated list of user names. These users are members of the group. Most Unix systems allow users to be members of several groups simultaneously. If your system does, then reading '`/dev/user`' will return those group-id numbers in `$5` through `$NF`. (Note that '`/dev/user`' is a `gawk` extension; see Section 6.7 [Special File Names in `gawk`], page 69.)

Here is what running `grcat` might produce:

```
$ grcat
⊣ wheel:*:0:arnold
⊣ nogroup:*:65534:
⊣ daemon:*:1:
⊣ kmem:*:2:
⊣ staff:*:10:arnold,miriam,andy
⊣ other:*:20:
...
```

Here are the functions for obtaining information from the group database. There are several, modeled after the C library functions of the same names.

```
# group.awk --- functions for dealing with the group file
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

BEGIN    \
{
    # Change to suit your system
    _gr_awklib = "/usr/local/libexec/awk/"
}

function _gr_init(    oldfs, oldrs, olddol0, grcat, n, a, i)
{
    if (_gr_inited)
        return

    oldfs = FS
    oldrs = RS
    olddol0 = $0
    FS = ":"
    RS = "\n"

    grcat = _gr_awklib "grcat"
    while ((grcat | getline) > 0) {
        if ($1 in _gr_byname)
            _gr_byname[$1] = _gr_byname[$1] "," $4
        else
            _gr_byname[$1] = $0
        if ($3 in _gr_bygid)
            _gr_bygid[$3] = _gr_bygid[$3] "," $4
        else
            _gr_bygid[$3] = $0

        n = split($4, a, "[ \t]*,[ \t]*")
        for (i = 1; i <= n; i++)
            if (a[i] in _gr_groupsbyuser)
                _gr_groupsbyuser[a[i]] = \
                    _gr_groupsbyuser[a[i]] " " $1
            else
                _gr_groupsbyuser[a[i]] = $1

        _gr_bycount[++_gr_count] = $0
    }
```

```
        close(grcat)
        _gr_count = 0
        _gr_inited++
        FS = oldfs
        RS = oldrs
        $0 = olddol0
    }
```

The `BEGIN` rule sets a private variable to the directory where `grcat` is stored. Since it is used to help out an `awk` library routine, we have chosen to put it in '`/usr/local/libexec/awk`'. You might want it to be in a different directory on your system.

These routines follow the same general outline as the user database routines (see Section 15.11 [Reading the User Database], page 189). The `_gr_inited` variable is used to ensure that the database is scanned no more than once. The `_gr_init` function first saves `FS`, `RS`, and `$0`, and then sets `FS` and `RS` to the correct values for scanning the group information.

The group information is stored is several associative arrays. The arrays are indexed by group name (`_gr_byname`), by group-id number (`_gr_bygid`), and by position in the database (`_gr_bycount`). There is an additional array indexed by user name (`_gr_groupsbyuser`), that is a space separated list of groups that each user belongs to.

Unlike the user database, it is possible to have multiple records in the database for the same group. This is common when a group has a large number of members. Such a pair of entries might look like:

```
tvpeople:*:101:johny,jay,arsenio
tvpeople:*:101:david,conan,tom,joan
```

For this reason, `_gr_init` looks to see if a group name or group-id number has already been seen. If it has, then the user names are simply concatenated onto the previous list of users. (There is actually a subtle problem with the code presented above. Suppose that the first time there were no names. This code adds the names with a leading comma. It also doesn't check that there is a `$4`.)

Finally, `_gr_init` closes the pipeline to `grcat`, restores `FS`, `RS`, and `$0`, initializes `_gr_count` to zero (it is used later), and makes `_gr_inited` non-zero.

```
function getgrnam(group)
{
    _gr_init()
    if (group in _gr_byname)
        return _gr_byname[group]
    return ""
}
```

The `getgrnam` function takes a group name as its argument, and if that group exists, it is returned. Otherwise, `getgrnam` returns the null string.

```
function getgrgid(gid)
{
    _gr_init()
    if (gid in _gr_bygid)
        return _gr_bygid[gid]
    return ""
}
```

The `getgrgid` function is similar, it takes a numeric group-id, and looks up the information associated with that group-id.

```
function getgruser(user)
{
    _gr_init()
    if (user in _gr_groupsbyuser)
        return _gr_groupsbyuser[user]
    return ""
}
```

The `getgruser` function does not have a C counterpart. It takes a user name, and returns the list of groups that have the user as a member.

```
function getgrent()
{
    _gr_init()
    if (++_gr_count in _gr_bycount)
        return _gr_bycount[_gr_count]
    return ""
}
```

The `getgrent` function steps through the database one entry at a time. It uses `_gr_count` to track its position in the list.

```
function endgrent()
{
    _gr_count = 0
}
```

`endgrent` resets `_gr_count` to zero so that `getgrent` can start over again.

As with the user database routines, each function calls `_gr_init` to initialize the arrays. Doing so only incurs the extra overhead of running `grcat` if these functions are used (as opposed to moving the body of `_gr_init` into a `BEGIN` rule).

Most of the work is in scanning the database and building the various associative arrays. The functions that the user calls are themselves very simple, relying on `awk`'s associative arrays to do work.

The `id` program in Section 16.1.3 [Printing Out User Information], page 210, uses these functions.

15.13 Naming Library Function Global Variables

Due to the way the `awk` language evolved, variables are either *global* (usable by the entire program), or *local* (usable just by a specific function). There is no intermediate state analogous to `static` variables in C.

Library functions often need to have global variables that they can use to preserve state information between calls to the function. For example, `getopt`'s variable `_opti` (see Section 15.10 [Processing Command Line Options], page 184), and the `_tm_months` array used by `mktime` (see Section 15.7 [Turning Dates Into Timestamps], page 175). Such variables are called *private*, since the only functions that need to use them are the ones in the library.

When writing a library function, you should try to choose names for your private variables so that they will not conflict with any variables used by either another library function or a user's main program. For example, a name like '`i`' or '`j`' is not a good choice, since user programs often use variable names like these for their own purposes.

The example programs shown in this chapter all start the names of their private variables with an underscore ('`_`'). Users generally don't use leading underscores in their variable names, so this convention immediately decreases the chances that the variable name will be accidentally shared with the user's program.

In addition, several of the library functions use a prefix that helps indicate what function or set of functions uses the variables. For example, `_tm_months` in `mktime` (see Section 15.7 [Turning Dates Into Timestamps], page 175), and `_pw_byname` in the user data base routines (see Section 15.11 [Reading the User Database], page 189). This convention is recommended, since it even further decreases the chance of inadvertent conflict among variable names. Note that this convention can be used equally well both for variable names and for private function names too.

While I could have re-written all the library routines to use this convention, I did not do so, in order to show how my own `awk` programming style has evolved, and to provide some basis for this discussion.

As a final note on variable naming, if a function makes global variables available for use by a main program, it is a good convention to start that variable's name with a capital letter. For example, `getopt`'s `Opterr` and `Optind` variables (see Section 15.10 [Processing Command Line Options], page 184). The leading capital letter indicates that it is global, while the fact that the variable name is not all capital letters indicates that the variable is not one of `awk`'s built-in variables, like `FS`.

It is also important that *all* variables in library functions that do not need to save state are in fact declared local. If this is not done, the variable could accidentally be used in the user's program, leading to bugs that are very difficult to track down.

```
function lib_func(x, y,    l1, l2)
{
    ...
    use variable some_var  # some_var could be local
    ...                     # but is not by oversight
}
```

A different convention, common in the Tcl community, is to use a single associative array to hold the values needed by the library function(s), or "package." This significantly decreases the number of actual global names in use. For example, the functions described in Section 15.11 [Reading the User Database], page 189, might have used `PW_data["inited"]`, `PW_data["total"]`, `PW_data["count"]` and `PW_data["awklib"]`, instead of `_pw_inited`, `_pw_awklib`, `_pw_total`, and `_pw_count`.

The conventions presented in this section are exactly that, conventions. You are not required to write your programs this way, we merely recommend that you do so.

16 Practical awk Programs

This chapter presents a potpourri of `awk` programs for your reading enjoyment. There are two sections. The first presents `awk` versions of several common POSIX utilities. The second is a grab-bag of interesting programs.

Many of these programs use the library functions presented in Chapter 15 [A Library of `awk` Functions], page 167.

16.1 Re-inventing Wheels for Fun and Profit

This section presents a number of POSIX utilities that are implemented in `awk`. Re-inventing these programs in `awk` is often enjoyable, since the algorithms can be very clearly expressed, and usually the code is very concise and simple. This is true because `awk` does so much for you.

It should be noted that these programs are not necessarily intended to replace the installed versions on your system. Instead, their purpose is to illustrate `awk` language programming for "real world" tasks.

The programs are presented in alphabetical order.

16.1.1 Cutting Out Fields and Columns

The `cut` utility selects, or "cuts," either characters or fields from its standard input and sends them to its standard output. `cut` can cut out either a list of characters, or a list of fields. By default, fields are separated by tabs, but you may supply a command line option to change the field *delimiter*, i.e. the field separator character. `cut`'s definition of fields is less general than `awk`'s.

A common use of `cut` might be to pull out just the login name of logged-on users from the output of `who`. For example, the following pipeline generates a sorted, unique list of the logged on users:

```
who | cut -c1-8 | sort | uniq
```

The options for `cut` are:

`-c` *list* Use *list* as the list of characters to cut out. Items within the list may be separated by commas, and ranges of characters can be separated with dashes. The list '`1-8,15,22-35`' specifies characters one through eight, 15, and 22 through 35.

`-f` *list* Use *list* as the list of fields to cut out.

`-d` *delim* Use *delim* as the field separator character instead of the tab character.

`-s` Suppress printing of lines that do not contain the field delimiter.

The `awk` implementation of `cut` uses the `getopt` library function (see Section 15.10 [Processing Command Line Options], page 184), and the `join` library function (see Section 15.6 [Merging an Array Into a String], page 174).

The program begins with a comment describing the options and a `usage` function which prints out a usage message and exits. `usage` is called if invalid arguments are supplied.

```
# cut.awk --- implement cut in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Options:
#    -f list        Cut fields
#    -d c           Field delimiter character
#    -c list        Cut characters
#
#    -s        Suppress lines without the delimiter character

function usage(    e1, e2)
{
    e1 = "usage: cut [-f list] [-d c] [-s] [files...]"
    e2 = "usage: cut [-c list] [files...]"
    print e1 > "/dev/stderr"
    print e2 > "/dev/stderr"
    exit 1
}
```

The variables `e1` and `e2` are used so that the function fits nicely on the page.

Next comes a `BEGIN` rule that parses the command line options. It sets `FS` to a single tab character, since that is `cut`'s default field separator. The output field separator is also set to be the same as the input field separator. Then `getopt` is used to step through the command line options. One or the other of the variables `by_fields` or `by_chars` is set to true, to indicate that processing should be done by fields or by characters respectively. When cutting by characters, the output field separator is set to the null string.

```
BEGIN    \
{
    FS = "\t"    # default
    OFS = FS
    while ((c = getopt(ARGC, ARGV, "sf:c:d:")) != -1) {
        if (c == "f") {
            by_fields = 1
            fieldlist = Optarg
        } else if (c == "c") {
            by_chars = 1
            fieldlist = Optarg
            OFS = ""
```

```
        } else if (c == "d") {
            if (length(Optarg) > 1) {
                printf("Using first character of %s" \
                " for delimiter\n", Optarg) > "/dev/stderr"
                Optarg = substr(Optarg, 1, 1)
            }
            FS = Optarg
            OFS = FS
            if (FS == " ")    # defeat awk semantics
                FS = "[ ]"
        } else if (c == "s")
            suppress++
        else
            usage()
    }

    for (i = 1; i < Optind; i++)
        ARGV[i] = ""
```

Special care is taken when the field delimiter is a space. Using `" "` (a single space) for the value of `FS` is incorrect—`awk` would separate fields with runs of spaces, tabs and/or newlines, and we want them to be separated with individual spaces. Also, note that after `getopt` is through, we have to clear out all the elements of `ARGV` from one to `Optind`, so that `awk` will not try to process the command line options as file names.

After dealing with the command line options, the program verifies that the options make sense. Only one or the other of '`-c`' and '`-f`' should be used, and both require a field list. Then either `set_fieldlist` or `set_charlist` is called to pull apart the list of fields or characters.

```
    if (by_fields && by_chars)
        usage()

    if (by_fields == 0 && by_chars == 0)
        by_fields = 1    # default

    if (fieldlist == "") {
        print "cut: needs list for -c or -f" > "/dev/stderr"
        exit 1
    }

    if (by_fields)
        set_fieldlist()
    else
        set_charlist()
}
```

Here is `set_fieldlist`. It first splits the field list apart at the commas, into an array. Then, for each element of the array, it looks to see if it is actually a range, and if so splits it apart. The range is verified to make sure the first number is smaller than the second. Each number in the list is added to the `flist` array, which simply lists the fields that will be printed. Normal field splitting is used. The program lets `awk` handle the job of doing the field splitting.

```
function set_fieldlist(        n, m, i, j, k, f, g)
{
    n = split(fieldlist, f, ",")
    j = 1    # index in flist
    for (i = 1; i <= n; i++) {
        if (index(f[i], "-") != 0) { # a range
            m = split(f[i], g, "-")
            if (m != 2 || g[1] >= g[2]) {
                printf("bad field list: %s\n",
                                f[i]) > "/dev/stderr"
                exit 1
            }
            for (k = g[1]; k <= g[2]; k++)
                flist[j++] = k
        } else
            flist[j++] = f[i]
    }
    nfields = j - 1
}
```

The `set_charlist` function is more complicated than `set_fieldlist`. The idea here is to use `gawk`'s `FIELDWIDTHS` variable (see Section 5.6 [Reading Fixed-width Data], page 49), which describes constant width input. When using a character list, that is exactly what we have.

Setting up `FIELDWIDTHS` is more complicated than simply listing the fields that need to be printed. We have to keep track of the fields to be printed, and also the intervening characters that have to be skipped. For example, suppose you wanted characters one through eight, 15, and 22 through 35. You would use '`-c 1-8,15,22-35`'. The necessary value for `FIELDWIDTHS` would be `"8 6 1 6 14"`. This gives us five fields, and what should be printed are `$1`, `$3`, and `$5`. The intermediate fields are "filler," stuff in between the desired data.

`flist` lists the fields to be printed, and `t` tracks the complete field list, including filler fields.

```
function set_charlist(    field, i, j, f, g, t,
                          filler, last, len)
{
    field = 1   # count total fields
```

```
        n = split(fieldlist, f, ",")
        j = 1       # index in flist
        for (i = 1; i <= n; i++) {
            if (index(f[i], "-") != 0) { # range
                m = split(f[i], g, "-")
                if (m != 2 || g[1] >= g[2]) {
                    printf("bad character list: %s\n",
                                    f[i]) > "/dev/stderr"
                    exit 1
                }
                len = g[2] - g[1] + 1
                if (g[1] > 1)  # compute length of filler
                    filler = g[1] - last - 1
                else
                    filler = 0
                if (filler)
                    t[field++] = filler
                t[field++] = len  # length of field
                last = g[2]
                flist[j++] = field - 1
            } else {
                if (f[i] > 1)
                    filler = f[i] - last - 1
                else
                    filler = 0
                if (filler)
                    t[field++] = filler
                t[field++] = 1
                last = f[i]
                flist[j++] = field - 1
            }
        }
        FIELDWIDTHS = join(t, 1, field - 1)
        nfields = j - 1
    }
```

Here is the rule that actually processes the data. If the '`-s`' option was given, then `suppress` will be true. The first `if` statement makes sure that the input record does have the field separator. If `cut` is processing fields, `suppress` is true, and the field separator character is not in the record, then the record is skipped.

If the record is valid, then at this point, `gawk` has split the data into fields, either using the character in `FS` or using fixed-length fields and `FIELDWIDTHS`. The loop goes through the list of fields that should be printed. If the cor-

responding field has data in it, it is printed. If the next field also has data, then the separator character is written out in between the fields.

```
{
    if (by_fields && suppress && $0 !~ FS)
        next

    for (i = 1; i <= nfields; i++) {
        if ($flist[i] != "") {
            printf "%s", $flist[i]
            if (i < nfields && $flist[i+1] != "")
                printf "%s", OFS
        }
    }
    print ""
}
```

This version of `cut` relies on `gawk`'s `FIELDWIDTHS` variable to do the character-based cutting. While it would be possible in other `awk` implementations to use `substr` (see Section 12.3 [Built-in Functions for String Manipulation], page 135), it would also be extremely painful to do so. The `FIELDWIDTHS` variable supplies an elegant solution to the problem of picking the input line apart by characters.

16.1.2 Searching for Regular Expressions in Files

The `egrep` utility searches files for patterns. It uses regular expressions that are almost identical to those available in `awk` (see Section 7.1.2 [Regular Expression Constants], page 75). It is used this way:

`egrep` [*options*] *'pattern' files* ...

The *pattern* is a regexp. In typical usage, the regexp is quoted to prevent the shell from expanding any of the special characters as file name wildcards. Normally, `egrep` prints the lines that matched. If multiple file names are provided on the command line, each output line is preceded by the name of the file and a colon.

The options are:

`-c` Print out a count of the lines that matched the pattern, instead of the lines themselves.

`-s` Be silent. No output is produced, and the exit value indicates whether or not the pattern was matched.

`-v` Invert the sense of the test. `egrep` prints the lines that do *not* match the pattern, and exits successfully if the pattern was not matched.

`-i` Ignore case distinctions in both the pattern and the input data.

`-l` Only print the names of the files that matched, not the lines that matched.

`-e` *pattern* Use *pattern* as the regexp to match. The purpose of the '`-e`' option is to allow patterns that start with a '`-`'.

This version uses the `getopt` library function (see Section 15.10 [Processing Command Line Options], page 184), and the file transition library program (see Section 15.9 [Noting Data File Boundaries], page 182).

The program begins with a descriptive comment, and then a `BEGIN` rule that processes the command line arguments with `getopt`. The '`-i`' (ignore case) option is particularly easy with `gawk`; we just use the `IGNORECASE` built in variable (see Chapter 10 [Built-in Variables], page 113).

```
# egrep.awk --- simulate egrep in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Options:
#    -c    count of lines
#    -s    silent - use exit value
#    -v    invert test, success if no match
#    -i    ignore case
#    -l    print filenames only
#    -e    argument is pattern

BEGIN {
    while ((c = getopt(ARGC, ARGV, "ce:svil")) != -1) {
        if (c == "c")
            count_only++
        else if (c == "s")
            no_print++
        else if (c == "v")
            invert++
        else if (c == "i")
            IGNORECASE = 1
        else if (c == "l")
            filenames_only++
        else if (c == "e")
            pattern = Optarg
        else
            usage()
    }
```

Next comes the code that handles the `egrep` specific behavior. If no pattern was supplied with '`-e`', the first non-option on the command line is used. The `awk` command line arguments up to `ARGV[Optind]` are cleared,

so that `awk` won't try to process them as files. If no files were specified, the standard input is used, and if multiple files were specified, we make sure to note this so that the file names can precede the matched lines in the output.

The last two lines are commented out, since they are not needed in `gawk`. They should be uncommented if you have to use another version of `awk`.

```
        if (pattern == "")
            pattern = ARGV[Optind++]

        for (i = 1; i < Optind; i++)
            ARGV[i] = ""
        if (Optind >= ARGC) {
            ARGV[1] = "-"
            ARGC = 2
        } else if (ARGC - Optind > 1)
            do_filenames++

    #    if (IGNORECASE)
    #        pattern = tolower(pattern)
    }
```

The next set of lines should be uncommented if you are not using `gawk`. This rule translates all the characters in the input line into lower-case if the '`-i`' option was specified. The rule is commented out since it is not necessary with `gawk`.

```
    #{
    #    if (IGNORECASE)
    #        $0 = tolower($0)
    #}
```

The `beginfile` function is called by the rule in '`ftrans.awk`' when each new file is processed. In this case, it is very simple; all it does is initialize a variable `fcount` to zero. `fcount` tracks how many lines in the current file matched the pattern.

```
    function beginfile(junk)
    {
        fcount = 0
    }
```

The `endfile` function is called after each file has been processed. It is used only when the user wants a count of the number of lines that matched. `no_print` will be true only if the exit status is desired. `count_only` will be true if line counts are desired. `egrep` will therefore only print line counts if printing and counting are enabled. The output format must be adjusted depending upon the number of files to be processed. Finally, `fcount` is added to `total`, so that we know how many lines altogether matched the pattern.

```
function endfile(file)
{
    if (! no_print && count_only)
        if (do_filenames)
            print file ":" fcount
        else
            print fcount

    total += fcount
}
```

This rule does most of the work of matching lines. The variable `matches` will be true if the line matched the pattern. If the user wants lines that did not match, the sense of the `matches` is inverted using the '`!`' operator. `fcount` is incremented with the value of `matches`, which will be either one or zero, depending upon a successful or unsuccessful match. If the line did not match, the `next` statement just moves on to the next record.

There are several optimizations for performance in the following few lines of code. If the user only wants exit status (`no_print` is true), and we don't have to count lines, then it is enough to know that one line in this file matched, and we can skip on to the next file with `nextfile`. Along similar lines, if we are only printing file names, and we don't need to count lines, we can print the file name, and then skip to the next file with `nextfile`.

Finally, each line is printed, with a leading filename and colon if necessary.

```
{
    matches = ($0 ~ pattern)
    if (invert)
        matches = ! matches

    fcount += matches    # 1 or 0

    if (! matches)
        next

    if (no_print && ! count_only)
        nextfile

    if (filenames_only && ! count_only) {
        print FILENAME
        nextfile
    }

    if (do_filenames && ! count_only)
        print FILENAME ":" $0
```

```
        else if (! count_only)
            print
    }
```

The `END` rule takes care of producing the correct exit status. If there were no matches, the exit status is one, otherwise it is zero.

```
    END    \
    {
        if (total == 0)
            exit 1
        exit 0
    }
```

The `usage` function prints a usage message in case of invalid options and then exits.

```
    function usage(    e)
    {
        e = "Usage: egrep [-csvil] [-e pat] [files ...]"
        print e > "/dev/stderr"
        exit 1
    }
```

The variable `e` is used so that the function fits nicely on the printed page.

Just a note on programming style. You may have noticed that the `END` rule uses backslash continuation, with the open brace on a line by itself. This is so that it more closely resembles the way functions are written. Many of the examples in this chapter use this style. You can decide for yourself if you like writing your `BEGIN` and `END` rules this way, or not.

16.1.3 Printing Out User Information

The `id` utility lists a user's real and effective user-id numbers, real and effective group-id numbers, and the user's group set, if any. `id` will only print the effective user-id and group-id if they are different from the real ones. If possible, `id` will also supply the corresponding user and group names. The output might look like this:

```
    $ id
    -| uid=2076(arnold) gid=10(staff) groups=10(staff),4(tty)
```

This information is exactly what is provided by `gawk`'s '`/dev/user`' special file (see Section 6.7 [Special File Names in `gawk`], page 69). However, the `id` utility provides a more palatable output than just a string of numbers.

Here is a simple version of `id` written in `awk`. It uses the user database library functions (see Section 15.11 [Reading the User Database], page 189), and the group database library functions (see Section 15.12 [Reading the Group Database], page 194).

The program is fairly straightforward. All the work is done in the `BEGIN` rule. The user and group id numbers are obtained from '`/dev/user`'. If there is no support for '`/dev/user`', the program gives up.

The code is repetitive. The entry in the user database for the real user-id number is split into parts at the ':'. The name is the first field. Similar code is used for the effective user-id number, and the group numbers.

```
# id.awk --- implement id in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# output is:
# uid=12(foo) euid=34(bar) gid=3(baz) \
#             egid=5(blat) groups=9(nine),2(two),1(one)

BEGIN    \
{
    if ((getline < "/dev/user") < 0) {
        err = "id: no /dev/user support - cannot run"
        print err > "/dev/stderr"
        exit 1
    }
    close("/dev/user")

    uid = $1
    euid = $2
    gid = $3
    egid = $4

    printf("uid=%d", uid)
    pw = getpwuid(uid)
    if (pw != "") {
        split(pw, a, ":")
        printf("(%s)", a[1])
    }
    if (euid != uid) {
        printf(" euid=%d", euid)
        pw = getpwuid(euid)
        if (pw != "") {
            split(pw, a, ":")
            printf("(%s)", a[1])
        }
    }
```

```
        printf(" gid=%d", gid)
        pw = getgrgid(gid)
        if (pw != "") {
            split(pw, a, ":")
            printf("(%s)", a[1])
        }

        if (egid != gid) {
            printf(" egid=%d", egid)
            pw = getgrgid(egid)
            if (pw != "") {
                split(pw, a, ":")
                printf("(%s)", a[1])
            }
        }

        if (NF > 4) {
            printf(" groups=");
            for (i = 5; i <= NF; i++) {
                printf("%d", $i)
                pw = getgrgid($i)
                if (pw != "") {
                    split(pw, a, ":")
                    printf("(%s)", a[1])
                }
                if (i < NF)
                    printf(",")
            }
        }
        print ""
    }
```

16.1.4 Splitting a Large File Into Pieces

The `split` program splits large text files into smaller pieces. By default, the output files are named '`xaa`', '`xab`', and so on. Each file has 1000 lines in it, with the likely exception of the last file. To change the number of lines in each file, you supply a number on the command line preceded with a minus, e.g., '`-500`' for files with 500 lines in them instead of 1000. To change the name of the output files to something like '`myfileaa`', '`myfileab`', and so on, you supply an additional argument that specifies the filename.

Here is a version of `split` in `awk`. It uses the `ord` and `chr` functions presented in Section 15.5 [Translating Between Characters and Numbers], page 172.

The program first sets its defaults, and then tests to make sure there are not too many arguments. It then looks at each argument in turn. The first argument could be a minus followed by a number. If it is, this happens to look like a negative number, so it is made positive, and that is the count of lines. The data file name is skipped over, and the final argument is used as the prefix for the output file names.

```
# split.awk --- do split in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# usage: split [-num] [file] [outname]

BEGIN {
    outfile = "x"    # default
    count = 1000
    if (ARGC > 4)
        usage()

    i = 1
    if (ARGV[i] ~ /^-[0-9]+$/) {
        count = -ARGV[i]
        ARGV[i] = ""
        i++
    }
    # test argv in case reading from stdin instead of file
    if (i in ARGV)
        i++    # skip data file name
    if (i in ARGV) {
        outfile = ARGV[i]
        ARGV[i] = ""
    }

    s1 = s2 = "a"
    out = (outfile s1 s2)
}
```

The next rule does most of the work. `tcount` (temporary count) tracks how many lines have been printed to the output file so far. If it is greater than `count`, it is time to close the current file and start a new one. `s1` and `s2` track the current suffixes for the file name. If they are both '`z`', the file is just too big. Otherwise, `s1` moves to the next letter in the alphabet and `s2` starts over again at '`a`'.

```
{
    if (++tcount > count) {
        close(out)
```

```
            if (s2 == "z") {
                if (s1 == "z") {
                    printf("split: %s is too large to split\n", \
                           FILENAME) > "/dev/stderr"
                    exit 1
                }
                s1 = chr(ord(s1) + 1)
                s2 = "a"
            } else
                s2 = chr(ord(s2) + 1)
            out = (outfile s1 s2)
            tcount = 1
        }
        print > out
    }
```

The `usage` function simply prints an error message and exits.

```
function usage(   e)
{
    e = "usage: split [-num] [file] [outname]"
    print e > "/dev/stderr"
    exit 1
}
```

The variable `e` is used so that the function fits nicely on the page.

This program is a bit sloppy; it relies on `awk` to close the last file for it automatically, instead of doing it in an `END` rule.

16.1.5 Duplicating Output Into Multiple Files

The `tee` program is known as a "pipe fitting." `tee` copies its standard input to its standard output, and also duplicates it to the files named on the command line. Its usage is:

```
tee [-a] file ...
```

The '`-a`' option tells `tee` to append to the named files, instead of truncating them and starting over.

The `BEGIN` rule first makes a copy of all the command line arguments, into an array named `copy`. `ARGV[0]` is not copied, since it is not needed. `tee` cannot use `ARGV` directly, since `awk` will attempt to process each file named in `ARGV` as input data.

If the first argument is '`-a`', then the flag variable `append` is set to true, and both `ARGV[1]` and `copy[1]` are deleted. If `ARGC` is less than two, then no file names were supplied, and `tee` prints a usage message and exits. Finally, `awk` is forced to read the standard input by setting `ARGV[1]` to `"-"`, and `ARGC` to two.

```
# tee.awk --- tee in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993
# Revised December 1995

BEGIN    \
{
    for (i = 1; i < ARGC; i++)
        copy[i] = ARGV[i]

    if (ARGV[1] == "-a") {
        append = 1
        delete ARGV[1]
        delete copy[1]
        ARGC--
    }
    if (ARGC < 2) {
        print "usage: tee [-a] file ..." > "/dev/stderr"
        exit 1
    }
    ARGV[1] = "-"
    ARGC = 2
}
```

The single rule does all the work. Since there is no pattern, it is executed for each line of input. The body of the rule simply prints the line into each file on the command line, and then to the standard output.

```
{
    # moving the if outside the loop makes it run faster
    if (append)
        for (i in copy)
            print >> copy[i]
    else
        for (i in copy)
            print > copy[i]
    print
}
```

It would have been possible to code the loop this way:

```
for (i in copy)
    if (append)
        print >> copy[i]
    else
        print > copy[i]
```

This is more concise, but it is also less efficient. The '`if`' is tested for each record and for each output file. By duplicating the loop body, the '`if`' is

only tested once for each input record. If there are N input records and M input files, the first method only executes N 'if' statements, while the second would execute $N*M$ 'if' statements.

Finally, the END rule cleans up, by closing all the output files.

```
END    \
{
    for (i in copy)
        close(copy[i])
}
```

16.1.6 Printing Non-duplicated Lines of Text

The uniq utility reads sorted lines of data on its standard input, and (by default) removes duplicate lines. In other words, only unique lines are printed, hence the name. uniq has a number of options. The usage is:

uniq [-udc [-*n*]] [+*n*] [*input file* [*output file*]]

The option meanings are:

-d Only print repeated lines.

-u Only print non-repeated lines.

-c Count lines. This option overrides '-d' and '-u'. Both repeated and non-repeated lines are counted.

-*n* Skip *n* fields before comparing lines. The definition of fields is similar to awk's default: non-whitespace characters separated by runs of spaces and/or tabs.

+*n* Skip *n* characters before comparing lines. Any fields specified with '-*n*' are skipped first.

input file Data is read from the input file named on the command line, instead of from the standard input.

output file The generated output is sent to the named output file, instead of to the standard output.

Normally uniq behaves as if both the '-d' and '-u' options had been provided.

Here is an awk implementation of uniq. It uses the getopt library function (see Section 15.10 [Processing Command Line Options], page 184), and the join library function (see Section 15.6 [Merging an Array Into a String], page 174).

The program begins with a usage function and then a brief outline of the options and their meanings in a comment.

The BEGIN rule deals with the command line arguments and options. It uses a trick to get getopt to handle options of the form '-25', treating such an option as the option letter '2' with an argument of '5'. If indeed

two or more digits were supplied (`Optarg` looks like a number), `Optarg` is concatenated with the option digit, and then result is added to zero to make it into a number. If there is only one digit in the option, then `Optarg` is not needed, and `Optind` must be decremented so that `getopt` will process it next time. This code is admittedly a bit tricky.

If no options were supplied, then the default is taken, to print both repeated and non-repeated lines. The output file, if provided, is assigned to `outputfile`. Earlier, `outputfile` was initialized to the standard output, '`/dev/stdout`'.

```
# uniq.awk --- do uniq in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

function usage(    e)
{
    e = "Usage: uniq [-udc [-n]] [+n] [ in [ out ]]"
    print e > "/dev/stderr"
    exit 1
}

# -c    count lines. overrides -d and -u
# -d    only repeated lines
# -u    only non-repeated lines
# -n    skip n fields
# +n    skip n characters, skip fields first

BEGIN   \
{
    count = 1
    outputfile = "/dev/stdout"
    opts = "udc0:1:2:3:4:5:6:7:8:9:"
    while ((c = getopt(ARGC, ARGV, opts)) != -1) {
        if (c == "u")
            non_repeated_only++
        else if (c == "d")
            repeated_only++
        else if (c == "c")
            do_count++
        else if (index("0123456789", c) != 0) {
            # getopt requires args to options
            # this messes us up for things like -5
            if (Optarg ~ /^[0-9]+$/)
                fcount = (c Optarg) + 0
```

```
            else {
                fcount = c + 0
                Optind--
            }
        } else
            usage()
    }

    if (ARGV[Optind] ~ /^\+[0-9]+$/) {
        charcount = substr(ARGV[Optind], 2) + 0
        Optind++
    }

    for (i = 1; i < Optind; i++)
        ARGV[i] = ""

    if (repeated_only == 0 && non_repeated_only == 0)
        repeated_only = non_repeated_only = 1

    if (ARGC - Optind == 2) {
        outputfile = ARGV[ARGC - 1]
        ARGV[ARGC - 1] = ""
    }
}
```

The following function, `are_equal`, compares the current line, `$0`, to the previous line, `last`. It handles skipping fields and characters.

If no field count and no character count were specified, `are_equal` simply returns one or zero depending upon the result of a simple string comparison of `last` and `$0`. Otherwise, things get more complicated.

If fields have to be skipped, each line is broken into an array using `split` (see Section 12.3 [Built-in Functions for String Manipulation], page 135), and then the desired fields are joined back into a line using `join`. The joined lines are stored in `clast` and `cline`. If no fields are skipped, `clast` and `cline` are set to `last` and `$0` respectively.

Finally, if characters are skipped, `substr` is used to strip off the leading `charcount` characters in `clast` and `cline`. The two strings are then compared, and `are_equal` returns the result.

```
function are_equal(    n, m, clast, cline, alast, aline)
{
    if (fcount == 0 && charcount == 0)
        return (last == $0)

    if (fcount > 0) {
        n = split(last, alast)
```

```
        m = split($0, aline)
        clast = join(alast, fcount+1, n)
        cline = join(aline, fcount+1, m)
    } else {
        clast = last
        cline = $0
    }
    if (charcount) {
        clast = substr(clast, charcount + 1)
        cline = substr(cline, charcount + 1)
    }

    return (clast == cline)
}
```

The following two rules are the body of the program. The first one is executed only for the very first line of data. It sets `last` equal to `$0`, so that subsequent lines of text have something to be compared to.

The second rule does the work. The variable `equal` will be one or zero depending upon the results of `are_equal`'s comparison. If `uniq` is counting repeated lines, then the `count` variable is incremented if the lines are equal. Otherwise the line is printed and `count` is reset, since the two lines are not equal.

If `uniq` is not counting, `count` is incremented if the lines are equal. Otherwise, if `uniq` is counting repeated lines, and more than one line has been seen, or if `uniq` is counting non-repeated lines, and only one line has been seen, then the line is printed, and `count` is reset.

Finally, similar logic is used in the `END` rule to print the final line of input data.

```
NR == 1 {
    last = $0
    next
}
{
    equal = are_equal()

    if (do_count) {    # overrides -d and -u
        if (equal)
            count++
        else {
            printf("%4d %s\n", count, last) > outputfile
            last = $0
            count = 1    # reset
        }
```

```
            next
        }

        if (equal)
            count++
        else {
            if ((repeated_only && count > 1) ||
                (non_repeated_only && count == 1))
                    print last > outputfile
            last = $0
            count = 1
        }
    }

    END {
        if (do_count)
            printf("%4d %s\n", count, last) > outputfile
        else if ((repeated_only && count > 1) ||
                 (non_repeated_only && count == 1))
            print last > outputfile
    }
```

16.1.7 Counting Things

The `wc` (word count) utility counts lines, words, and characters in one or more input files. Its usage is:

`wc` [`-lwc`] [*files* `...`]

If no files are specified on the command line, `wc` reads its standard input. If there are multiple files, it will also print total counts for all the files. The options and their meanings are:

`-l` Only count lines.

`-w` Only count words. A "word" is a contiguous sequence of non-whitespace characters, separated by spaces and/or tabs. Happily, this is the normal way `awk` separates fields in its input data.

`-c` Only count characters.

Implementing `wc` in `awk` is particularly elegant, since `awk` does a lot of the work for us; it splits lines into words (i.e. fields) and counts them, it counts lines (i.e. records) for us, and it can easily tell us how long a line is.

This version uses the `getopt` library function (see Section 15.10 [Processing Command Line Options], page 184), and the file transition functions (see Section 15.9 [Noting Data File Boundaries], page 182).

This version has one major difference from traditional versions of `wc`. Our version always prints the counts in the order lines, words, and characters.

Traditional versions note the order of the '`-l`', '`-w`', and '`-c`' options on the command line, and print the counts in that order.

The `BEGIN` rule does the argument processing. The variable `print_total` will be true if more than one file was named on the command line.

```
# wc.awk --- count lines, words, characters
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Options:
#    -l    only count lines
#    -w    only count words
#    -c    only count characters
#
# Default is to count lines, words, characters

BEGIN {
    # let getopt print a message about
    # invalid options. we ignore them
    while ((c = getopt(ARGC, ARGV, "lwc")) != -1) {
        if (c == "l")
            do_lines = 1
        else if (c == "w")
            do_words = 1
        else if (c == "c")
            do_chars = 1
    }
    for (i = 1; i < Optind; i++)
        ARGV[i] = ""

    # if no options, do all
    if (! do_lines && ! do_words && ! do_chars)
        do_lines = do_words = do_chars = 1

    print_total = (ARGC - i > 2)
}
```

The `beginfile` function is simple; it just resets the counts of lines, words, and characters to zero, and saves the current file name in `fname`.

The `endfile` function adds the current file's numbers to the running totals of lines, words, and characters. It then prints out those numbers for the file that was just read. It relies on `beginfile` to reset the numbers for the following data file.

```
function beginfile(file) {
    chars = lines = words = 0
    fname = FILENAME
}

function endfile(file)
{
    tchars += chars
    tlines += lines
    twords += words
    if (do_lines)
        printf "\t%d", lines
    if (do_words)
        printf "\t%d", words
    if (do_chars)
        printf "\t%d", chars
    printf "\t%s\n", fname
}
```

There is one rule that is executed for each line. It adds the length of the record to `chars`. It has to add one, since the newline character separating records (the value of `RS`) is not part of the record itself. `lines` is incremented for each line read, and `words` is incremented by the value of `NF`, the number of "words" on this line.[1]

Finally, the `END` rule simply prints the totals for all the files.

```
# do per line
{
    chars += length($0) + 1    # get newline
    lines++
    words += NF
}

END {
    if (print_total) {
        if (do_lines)
            printf "\t%d", tlines
        if (do_words)
            printf "\t%d", twords
        if (do_chars)
            printf "\t%d", tchars
        print "\ttotal"
    }
}
```

[1] Examine the code in Section 15.9 [Noting Data File Boundaries], page 182. Why must `wc` use a separate `lines` variable, instead of using the value of `FNR` in `endfile`?

16.2 A Grab Bag of awk Programs

This section is a large "grab bag" of miscellaneous programs. We hope you find them both interesting and enjoyable.

16.2.1 Finding Duplicated Words in a Document

A common error when writing large amounts of prose is to accidentally duplicate words. Often you will see this in text as something like "the the program does the following ...." When the text is on-line, often the duplicated words occur at the end of one line and the beginning of another, making them very difficult to spot.

This program, '`dupword.awk`', scans through a file one line at a time, and looks for adjacent occurrences of the same word. It also saves the last word on a line (in the variable `prev`) for comparison with the first word on the next line.

The first two statements make sure that the line is all lower-case, so that, for example, "The" and "the" compare equal to each other. The second statement removes all non-alphanumeric and non-whitespace characters from the line, so that punctuation does not affect the comparison either. This sometimes leads to reports of duplicated words that really are different, but this is unusual.

```
# dupword --- find duplicate words in text
# Arnold Robbins, arnold@gnu.org, Public Domain
# December 1991

{
    $0 = tolower($0)
    gsub(/[^A-Za-z0-9 \t]/, "");
    if ($1 == prev)
        printf("%s:%d: duplicate %s\n",
            FILENAME, FNR, $1)
    for (i = 2; i <= NF; i++)
        if ($i == $(i-1))
            printf("%s:%d: duplicate %s\n",
                FILENAME, FNR, $i)
    prev = $NF
}
```

16.2.2 An Alarm Clock Program

The following program is a simple "alarm clock" program. You give it a time of day, and an optional message. At the given time, it prints the message on the standard output. In addition, you can give it the number of times to repeat the message, and also a delay between repetitions.

This program uses the `gettimeofday` function from Section 15.8 [Managing the Time of Day], page 180.

All the work is done in the `BEGIN` rule. The first part is argument checking and setting of defaults; the delay, the count, and the message to print. If the user supplied a message, but it does not contain the ASCII BEL character (known as the "alert" character, '`\a`'), then it is added to the message. (On many systems, printing the ASCII BEL generates some sort of audible alert. Thus, when the alarm goes off, the system calls attention to itself, in case the user is not looking at their computer or terminal.)

```
# alarm --- set an alarm
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# usage: alarm time [ "message" [ count [ delay ] ] ]

BEGIN    \
{
    # Initial argument sanity checking
    usage1 = "usage: alarm time ['message' [count [delay]]]"
    usage2 = sprintf("\t(%s) time ::= hh:mm", ARGV[1])

    if (ARGC < 2) {
        print usage > "/dev/stderr"
        exit 1
    } else if (ARGC == 5) {
        delay = ARGV[4] + 0
        count = ARGV[3] + 0
        message = ARGV[2]
    } else if (ARGC == 4) {
        count = ARGV[3] + 0
        message = ARGV[2]
    } else if (ARGC == 3) {
        message = ARGV[2]
    } else if (ARGV[1] !~ /[0-9]?[0-9]:[0-9][0-9]/) {
        print usage1 > "/dev/stderr"
        print usage2 > "/dev/stderr"
        exit 1
    }

    # set defaults for once we reach the desired time
    if (delay == 0)
        delay = 180    # 3 minutes
    if (count == 0)
        count = 5
```

```
if (message == "")
    message = sprintf("\aIt is now %s!\a", ARGV[1])
else if (index(message, "\a") == 0)
    message = "\a" message "\a"
```

The next section of code turns the alarm time into hours and minutes, and converts it if necessary to a 24-hour clock. Then it turns that time into a count of the seconds since midnight. Next it turns the current time into a count of seconds since midnight. The difference between the two is how long to wait before setting off the alarm.

```
# split up dest time
split(ARGV[1], atime, ":")
hour = atime[1] + 0    # force numeric
minute = atime[2] + 0  # force numeric

# get current broken down time
gettimeofday(now)

# if time given is 12-hour hours and it's after that
# hour, e.g., 'alarm 5:30' at 9 a.m. means 5:30 p.m.,
# then add 12 to real hour
if (hour < 12 && now["hour"] > hour)
    hour += 12

# set target time in seconds since midnight
target = (hour * 60 * 60) + (minute * 60)

# get current time in seconds since midnight
current = (now["hour"] * 60 * 60) + \
           (now["minute"] * 60) + now["second"]

# how long to sleep for
naptime = target - current
if (naptime <= 0) {
    print "time is in the past!" > "/dev/stderr"
    exit 1
}
```

Finally, the program uses the `system` function (see Section 12.4 [Built-in Functions for Input/Output], page 143) to call the `sleep` utility. The `sleep` utility simply pauses for the given number of seconds. If the exit status is not zero, the program assumes that `sleep` was interrupted, and exits. If `sleep` exited with an OK status (zero), then the program prints the message in a loop, again using `sleep` to delay for however many seconds are necessary.

```
        # zzzzzz..... go away if interrupted
        if (system(sprintf("sleep %d", naptime)) != 0)
            exit 1

        # time to notify!
        command = sprintf("sleep %d", delay)
        for (i = 1; i <= count; i++) {
            print message
            # if sleep command interrupted, go away
            if (system(command) != 0)
                break
        }

        exit 0
    }
```

16.2.3 Transliterating Characters

The system `tr` utility transliterates characters. For example, it is often used to map upper-case letters into lower-case, for further processing.

generate data `|` `tr '[A-Z]' '[a-z]'` `|` *process data* `...`

You give `tr` two lists of characters enclosed in square brackets. Usually, the lists are quoted to keep the shell from attempting to do a filename expansion.[2] When processing the input, the first character in the first list is replaced with the first character in the second list, the second character in the first list is replaced with the second character in the second list, and so on. If there are more characters in the "from" list than in the "to" list, the last character of the "to" list is used for the remaining characters in the "from" list.

Some time ago, a user proposed to us that we add a transliteration function to `gawk`. Being opposed to "creeping featurism," I wrote the following program to prove that character transliteration could be done with a user-level function. This program is not as complete as the system `tr` utility, but it will do most of the job.

The `translate` program demonstrates one of the few weaknesses of standard `awk`: dealing with individual characters is very painful, requiring repeated use of the `substr`, `index`, and `gsub` built-in functions (see Section 12.3 [Built-in Functions for String Manipulation], page 135).[3]

There are two functions. The first, `stranslate`, takes three arguments.

---

[2] On older, non-POSIX systems, `tr` often does not require that the lists be enclosed in square brackets and quoted. This is a feature.

[3] This program was written before `gawk` acquired the ability to split each character in a string into separate array elements. How might you use this new feature to simplify the program?

`from` A list of characters to translate from.

`to` A list of characters to translate to.

`target` The string to do the translation on.

Associative arrays make the translation part fairly easy. `t_ar` holds the "to" characters, indexed by the "from" characters. Then a simple loop goes through `from`, one character at a time. For each character in `from`, if the character appears in `target`, `gsub` is used to change it to the corresponding `to` character.

The `translate` function simply calls `stranslate` using `$0` as the target. The main program sets two global variables, `FROM` and `TO`, from the command line, and then changes `ARGV` so that `awk` will read from the standard input.

Finally, the processing rule simply calls `translate` for each record.

```
# translate --- do tr like stuff
# Arnold Robbins, arnold@gnu.org, Public Domain
# August 1989

# bugs: does not handle things like: tr A-Z a-z, it has
# to be spelled out. However, if 'to' is shorter than 'from',
# the last character in 'to' is used for the rest of 'from'.

function stranslate(from, to, target,     lf, lt, t_ar, i, c)
{
    lf = length(from)
    lt = length(to)
    for (i = 1; i <= lt; i++)
        t_ar[substr(from, i, 1)] = substr(to, i, 1)
    if (lt < lf)
        for (; i <= lf; i++)
            t_ar[substr(from, i, 1)] = substr(to, lt, 1)
    for (i = 1; i <= lf; i++) {
        c = substr(from, i, 1)
        if (index(target, c) > 0)
            gsub(c, t_ar[c], target)
    }
    return target
}

function translate(from, to)
{
    return $0 = stranslate(from, to, $0)
}
```

```
# main program
BEGIN {
    if (ARGC < 3) {
        print "usage: translate from to" > "/dev/stderr"
        exit
    }
    FROM = ARGV[1]
    TO = ARGV[2]
    ARGC = 2
    ARGV[1] = "-"
}

{
    translate(FROM, TO)
    print
}
```

While it is possible to do character transliteration in a user-level function, it is not necessarily efficient, and we started to consider adding a built-in function. However, shortly after writing this program, we learned that the System V Release 4 `awk` had added the `toupper` and `tolower` functions. These functions handle the vast majority of the cases where character transliteration is necessary, and so we chose to simply add those functions to `gawk` as well, and then leave well enough alone.

An obvious improvement to this program would be to set up the `t_ar` array only once, in a `BEGIN` rule. However, this assumes that the "from" and "to" lists will never change throughout the lifetime of the program.

16.2.4 Printing Mailing Labels

Here is a "real world"[4] program. This script reads lists of names and addresses, and generates mailing labels. Each page of labels has 20 labels on it, two across and ten down. The addresses are guaranteed to be no more than five lines of data. Each address is separated from the next by a blank line.

The basic idea is to read 20 labels worth of data. Each line of each label is stored in the `line` array. The single rule takes care of filling the `line` array and printing the page when 20 labels have been read.

The `BEGIN` rule simply sets `RS` to the empty string, so that `awk` will split records at blank lines (see Section 5.1 [How Input is Split into Records], page 37). It sets `MAXLINES` to 100, since `MAXLINE` is the maximum number of lines on the page (20 * 5 = 100).

Most of the work is done in the `printpage` function. The label lines are stored sequentially in the `line` array. But they have to be printed

[4] "Real world" is defined as "a program actually used to get something done."

horizontally; `line[1]` next to `line[6]`, `line[2]` next to `line[7]`, and so on. Two loops are used to accomplish this. The outer loop, controlled by `i`, steps through every 10 lines of data; this is each row of labels. The inner loop, controlled by `j`, goes through the lines within the row. As `j` goes from zero to four, '`i+j`' is the `j`'th line in the row, and '`i+j+5`' is the entry next to it. The output ends up looking something like this:

```
line 1          line 6
line 2          line 7
line 3          line 8
line 4          line 9
line 5          line 10
```

As a final note, at lines 21 and 61, an extra blank line is printed, to keep the output lined up on the labels. This is dependent on the particular brand of labels in use when the program was written. You will also note that there are two blank lines at the top and two blank lines at the bottom.

The `END` rule arranges to flush the final page of labels; there may not have been an even multiple of 20 labels in the data.

```
# labels.awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# June 1992

# Program to print labels.  Each label is 5 lines of data
# that may have blank lines.  The label sheets have 2
# blank lines at the top and 2 at the bottom.

BEGIN    { RS = "" ; MAXLINES = 100 }

function printpage(    i, j)
{
    if (Nlines <= 0)
        return

    printf "\n\n"        # header

    for (i = 1; i <= Nlines; i += 10) {
        if (i == 21 || i == 61)
            print ""
        for (j = 0; j < 5; j++) {
            if (i + j > MAXLINES)
                break
            printf "   %-41s %s\n", line[i+j], line[i+j+5]
        }
        print ""
    }
```

```
    printf "\n\n"        # footer

    for (i in line)
        line[i] = ""
}

# main rule
{
    if (Count >= 20) {
        printpage()
        Count = 0
        Nlines = 0
    }
    n = split($0, a, "\n")
    for (i = 1; i <= n; i++)
        line[++Nlines] = a[i]
    for (; i <= 5; i++)
        line[++Nlines] = ""
    Count++
}

END    \
{
    printpage()
}
```

16.2.5 Generating Word Usage Counts

The following `awk` program prints the number of occurrences of each word in its input. It illustrates the associative nature of `awk` arrays by using strings as subscripts. It also demonstrates the '`for` *x* `in` *array*' construction. Finally, it shows how `awk` can be used in conjunction with other utility programs to do a useful task of some complexity with a minimum of effort. Some explanations follow the program listing.

```
awk '
# Print list of word frequencies
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}
```

```
END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}'
```

The first thing to notice about this program is that it has two rules. The first rule, because it has an empty pattern, is executed on every line of the input. It uses `awk`'s field-accessing mechanism (see Section 5.2 [Examining Fields], page 40) to pick out the individual words from the line, and the built-in variable `NF` (see Chapter 10 [Built-in Variables], page 113) to know how many fields are available.

For each input word, an element of the array `freq` is incremented to reflect that the word has been seen an additional time.

The second rule, because it has the pattern `END`, is not executed until the input has been exhausted. It prints out the contents of the `freq` table that has been built up inside the first action.

This program has several problems that would prevent it from being useful by itself on real text files:

- Words are detected using the `awk` convention that fields are separated by whitespace and that other characters in the input (except newlines) don't have any special meaning to `awk`. This means that punctuation characters count as part of words.
- The `awk` language considers upper- and lower-case characters to be distinct. Therefore, '`bartender`' and '`Bartender`' are not treated as the same word. This is undesirable since, in normal text, words are capitalized if they begin sentences, and a frequency analyzer should not be sensitive to capitalization.
- The output does not come out in any useful order. You're more likely to be interested in which words occur most frequently, or having an alphabetized table of how frequently each word occurs.

The way to solve these problems is to use some of the more advanced features of the `awk` language. First, we use `tolower` to remove case distinctions. Next, we use `gsub` to remove punctuation characters. Finally, we use the system `sort` utility to process the output of the `awk` script. Here is the new version of the program:

```
# Print list of word frequencies
{
    $0 = tolower($0)    # remove case distinctions
    gsub(/[^a-z0-9_ \t]/, "", $0)  # remove punctuation
    for (i = 1; i <= NF; i++)
        freq[$i]++
}
```

```
END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}
```

Assuming we have saved this program in a file named '`wordfreq.awk`', and that the data is in '`file1`', the following pipeline

```
awk -f wordfreq.awk file1 | sort +1 -nr
```

produces a table of the words appearing in '`file1`' in order of decreasing frequency.

The `awk` program suitably massages the data and produces a word frequency table, which is not ordered.

The `awk` script's output is then sorted by the `sort` utility and printed on the terminal. The options given to `sort` in this example specify to sort using the second field of each input line (skipping one field), that the sort keys should be treated as numeric quantities (otherwise '`15`' would come before '`5`'), and that the sorting should be done in descending (reverse) order.

We could have even done the `sort` from within the program, by changing the `END` action to:

```
END {
    sort = "sort +1 -nr"
    for (word in freq)
        printf "%s\t%d\n", word, freq[word] | sort
    close(sort)
}
```

You would have to use this way of sorting on systems that do not have true pipes.

See the general operating system documentation for more information on how to use the `sort` program.

16.2.6 Removing Duplicates from Unsorted Text

The `uniq` program (see Section 16.1.6 [Printing Non-duplicated Lines of Text], page 216), removes duplicate lines from *sorted* data.

Suppose, however, you need to remove duplicate lines from a data file, but that you wish to preserve the order the lines are in? A good example of this might be a shell history file. The history file keeps a copy of all the commands you have entered, and it is not unusual to repeat a command several times in a row. Occasionally you might wish to compact the history by removing duplicate entries. Yet it is desirable to maintain the order of the original commands.

This simple program does the job. It uses two arrays. The `data` array is indexed by the text of each line. For each line, `data[$0]` is incremented.

If a particular line has not been seen before, then `data[$0]` will be zero. In that case, the text of the line is stored in `lines[count]`. Each element of `lines` is a unique command, and the indices of `lines` indicate the order in which those lines were encountered. The `END` rule simply prints out the lines, in order.

```
# histsort.awk --- compact a shell history file
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Thanks to Byron Rakitzis for the general idea
{
    if (data[$0]++ == 0)
        lines[++count] = $0
}

END {
    for (i = 1; i <= count; i++)
        print lines[i]
}
```

This program also provides a foundation for generating other useful information. For example, using the following `print` satement in the `END` rule would indicate how often a particular command was used.

```
print data[lines[i]], lines[i]
```

This works because `data[$0]` was incremented each time a line was seen.

16.2.7 Extracting Programs from Texinfo Source Files

Both this chapter and the previous chapter (Chapter 15 [A Library of `awk` Functions], page 167), present a large number of `awk` programs. If you wish to experiment with these programs, it is tedious to have to type them in by hand. Here we present a program that can extract parts of a Texinfo input file into separate files.

This book is written in Texinfo, the GNU project's document formatting language. A single Texinfo source file can be used to produce both printed and on-line documentation. Texinfo is fully documented in *Texinfo—The GNU Documentation Format*, available from the Free Software Foundation.

For our purposes, it is enough to know three things about Texinfo input files.

- The "at" symbol, '`@`', is special in Texinfo, much like '\' in C or `awk`. Literal '`@`' symbols are represented in Texinfo source files as '`@@`'.
- Comments start with either '`@c`' or '`@comment`'. The file extraction program will work by using special comments that start at the beginning of a line.

- Example text that should not be split across a page boundary is bracketed between lines containing '`@group`' and '`@end group`' commands.

The following program, '`extract.awk`', reads through a Texinfo source file, and does two things, based on the special comments. Upon seeing '`@c system ...`', it runs a command, by extracting the command text from the control line and passing it on to the `system` function (see Section 12.4 [Built-in Functions for Input/Output], page 143). Upon seeing '`@c file` *filename*', each subsequent line is sent to the file *filename*, until '`@c endfile`' is encountered. The rules in '`extract.awk`' will match either '`@c`' or '`@comment`' by letting the '`omment`' part be optional. Lines containing '`@group`' and '`@end group`' are simply removed. '`extract.awk`' uses the `join` library function (see Section 15.6 [Merging an Array Into a String], page 174).

The example programs in the on-line Texinfo source for *Effective AWK Programming* ('`gawk.texi`') have all been bracketed inside '`file`', and '`endfile`' lines. The `gawk` distribution uses a copy of '`extract.awk`' to extract the sample programs and install many of them in a standard directory, where `gawk` can find them. The Texinfo file looks something like this:

```
...
This program has a @code{BEGIN} block,
which prints a nice message:

@example
@c file examples/messages.awk
BEGIN @{ print "Don't panic!" @}
@c end file
@end example

It also prints some final advice:

@example
@c file examples/messages.awk
END @{ print "Always avoid bored archeologists!" @}
@c end file
@end example
...
```

'`extract.awk`' begins by setting `IGNORECASE` to one, so that mixed upper-case and lower-case letters in the directives won't matter.

The first rule handles calling `system`, checking that a command was given (`NF` is at least three), and also checking that the command exited with a zero exit status, signifying OK.

```
# extract.awk --- extract files and run programs
#                 from texinfo files
# Arnold Robbins, arnold@gnu.org, Public Domain, May 1993
```

```
BEGIN    { IGNORECASE = 1 }

/^@c(omment)?[ \t]+system/    \
{
    if (NF < 3) {
        e = (FILENAME ":" FNR)
        e = (e  ": badly formed `system' line")
        print e > "/dev/stderr"
        next
    }
    $1 = ""
    $2 = ""
    stat = system($0)
    if (stat != 0) {
        e = (FILENAME ":" FNR)
        e = (e ": warning: system returned " stat)
        print e > "/dev/stderr"
    }
}
```

The variable `e` is used so that the function fits nicely on the page.

The second rule handles moving data into files. It verifies that a file name was given in the directive. If the file named is not the current file, then the current file is closed. This means that an '`@c endfile`' was not given for that file. (We should probably print a diagnostic in this case, although at the moment we do not.)

The '`for`' loop does the work. It reads lines using `getline` (see Section 5.8 [Explicit Input with `getline`], page 53). For an unexpected end of file, it calls the `unexpected_eof` function. If the line is an "endfile" line, then it breaks out of the loop. If the line is an '`@group`' or '`@end group`' line, then it ignores it, and goes on to the next line. (These Texinfo control lines keep blocks of code together on one page; unfortunately, TEX isn't always smart enough to do things exactly right, and we have to give it some advice.)

Most of the work is in the following few lines. If the line has no '`@`' symbols, it can be printed directly. Otherwise, each leading '`@`' must be stripped off.

To remove the '`@`' symbols, the line is split into separate elements of the array `a`, using the `split` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135). Each element of `a` that is empty indicates two successive '`@`' symbols in the original line. For each two empty elements ('`@@`' in the original file), we have to add back in a single '`@`' symbol.

When the processing of the array is finished, `join` is called with the value of `SUBSEP`, to rejoin the pieces back into a single line. That line is then printed to the output file.

```
/^@c(omment)?[ \t]+file/    \
{
    if (NF != 3) {
        e = (FILENAME ":" FNR ": badly formed `file' line")
        print e > "/dev/stderr"
        next
    }
    if ($3 != curfile) {
        if (curfile != "")
            close(curfile)
        curfile = $3
    }

    for (;;) {
        if ((getline line) <= 0)
            unexpected_eof()
        if (line ~ /^@c(omment)?[ \t]+endfile/)
            break
        else if (line ~ /^@(end[ \t]+)?group/)
            continue
        if (index(line, "@") == 0) {
            print line > curfile
            continue
        }
        n = split(line, a, "@")
        # if a[1] == "", means leading @,
        # don't add one back in.
        for (i = 2; i <= n; i++) {
            if (a[i] == "") { # was an @@
                a[i] = "@"
                if (a[i+1] == "")
                    i++
            }
        }
        print join(a, 1, n, SUBSEP) > curfile
    }
}
```

An important thing to note is the use of the '`>`' redirection. Output done with '`>`' only opens the file once; it stays open and subsequent output is appended to the file (see Section 6.6 [Redirecting Output of `print` and `printf`], page 67). This allows us to easily mix program text and explanatory prose for the same sample source file (as has been done here!) without any hassle. The file is only closed when a new data file name is encountered, or at the end of the input file.

Finally, the function `unexpected_eof` prints an appropriate error message and then exits.

The `END` rule handles the final cleanup, closing the open file.

```
function unexpected_eof()
{
    printf("%s:%d: unexpected EOF or error\n", \
        FILENAME, FNR) > "/dev/stderr"
    exit 1
}

END {
    if (curfile)
        close(curfile)
}
```

16.2.8 A Simple Stream Editor

The `sed` utility is a "stream editor," a program that reads a stream of data, makes changes to it, and passes the modified data on. It is often used to make global changes to a large file, or to a stream of data generated by a pipeline of commands.

While `sed` is a complicated program in its own right, its most common use is to perform global substitutions in the middle of a pipeline:

```
command1 < orig.data | sed 's/old/new/g' | command2 > result
```

Here, the '`s/old/new/g`' tells `sed` to look for the regexp '`old`' on each input line, and replace it with the text '`new`', globally (i.e. all the occurrences on a line). This is similar to `awk`'s `gsub` function (see Section 12.3 [Built-in Functions for String Manipulation], page 135).

The following program, '`awksed.awk`', accepts at least two command line arguments; the pattern to look for and the text to replace it with. Any additional arguments are treated as data file names to process. If none are provided, the standard input is used.

```
# awksed.awk --- do s/foo/bar/g using just print
#     Thanks to Michael Brennan for the idea

# Arnold Robbins, arnold@gnu.org, Public Domain
# August 1995

function usage()
{
    print "usage: awksed pat repl [files...]" > "/dev/stderr"
    exit 1
}
```

```
BEGIN {
    # validate arguments
    if (ARGC < 3)
        usage()

    RS = ARGV[1]
    ORS = ARGV[2]

    # don't use arguments as files
    ARGV[1] = ARGV[2] = ""
}

# look ma, no hands!
{
    if (RT == "")
        printf "%s", $0
    else
        print
}
```

The program relies on `gawk`'s ability to have `RS` be a regexp and on the setting of `RT` to the actual text that terminated the record (see Section 5.1 [How Input is Split into Records], page 37).

The idea is to have `RS` be the pattern to look for. `gawk` will automatically set `$0` to the text between matches of the pattern. This is text that we wish to keep, unmodified. Then, by setting `ORS` to the replacement text, a simple `print` statement will output the text we wish to keep, followed by the replacement text.

There is one wrinkle to this scheme, which is what to do if the last record doesn't end with text that matches `RS`? Using a `print` statement unconditionally prints the replacement text, which is not correct.

However, if the file did not end in text that matches `RS`, `RT` will be set to the null string. In this case, we can print `$0` using `printf` (see Section 6.5 [Using `printf` Statements for Fancier Printing], page 62).

The `BEGIN` rule handles the setup, checking for the right number of arguments, and calling `usage` if there is a problem. Then it sets `RS` and `ORS` from the command line arguments, and sets `ARGV[1]` and `ARGV[2]` to the null string, so that they will not be treated as file names (see Section 10.3 [Using `ARGC` and `ARGV`], page 117).

The `usage` function prints an error message and exits.

Finally, the single rule handles the printing scheme outlined above, using `print` or `printf` as appropriate, depending upon the value of `RT`.

16.2.9 An Easy Way to Use Library Functions

Using library functions in `awk` can be very beneficial. It encourages code re-use and the writing of general functions. Programs are smaller, and therefore clearer. However, using library functions is only easy when writing `awk` programs; it is painful when running them, requiring multiple '`-f`' options. If `gawk` is unavailable, then so too is the `AWKPATH` environment variable and the ability to put `awk` functions into a library directory (see Section 14.1 [Command Line Options], page 159).

It would be nice to be able to write programs like so:

```
# library functions
@include getopt.awk
@include join.awk
...

# main program
BEGIN {
    while ((c = getopt(ARGC, ARGV, "a:b:cde")) != -1)
        ...
    ...
}
```

The following program, '`igawk.sh`', provides this service. It simulates `gawk`'s searching of the `AWKPATH` variable, and also allows *nested* includes; i.e. a file that has been included with '`@include`' can contain further '`@include`' statements. `igawk` will make an effort to only include files once, so that nested includes don't accidentally include a library function twice.

`igawk` should behave externally just like `gawk`. This means it should accept all of `gawk`'s command line arguments, including the ability to have multiple source files specified via '`-f`', and the ability to mix command line and library source files.

The program is written using the POSIX Shell (`sh`) command language. The way the program works is as follows:

1. Loop through the arguments, saving anything that doesn't represent `awk` source code for later, when the expanded program is run.
2. For any arguments that do represent `awk` text, put the arguments into a temporary file that will be expanded. There are two cases.
   a. Literal text, provided with '`--source`' or '`--source=`'. This text is just echoed directly. The `echo` program will automatically supply a trailing newline.
   b. File names provided with '`-f`'. We use a neat trick, and echo '`@include` *filename*' into the temporary file. Since the file inclusion program will work the way `gawk` does, this will get the text of the file included into the program at the correct point.

3. Run an `awk` program (naturally) over the temporary file to expand '`@include`' statements. The expanded program is placed in a second temporary file.
4. Run the expanded program with `gawk` and any other original command line arguments that the user supplied (such as the data file names).

The initial part of the program turns on shell tracing if the first argument was '`debug`'. Otherwise, a shell `trap` statement arranges to clean up any temporary files on program exit or upon an interrupt.

The next part loops through all the command line arguments. There are several cases of interest.

`--` This ends the arguments to `igawk`. Anything else should be passed on to the user's `awk` program without being evaluated.

`-W` This indicates that the next option is specific to `gawk`. To make argument processing easier, the '`-W`' is appended to the front of the remaining arguments and the loop continues. (This is an `sh` programming trick. Don't worry about it if you are not familiar with `sh`.)

`-v`
`-F` These are saved and passed on to `gawk`.

`-f`
`--file`
`--file=`
`-Wfile=` The file name is saved to the temporary file '`/tmp/ig.s.$$`' with an '`@include`' statement. The `sed` utility is used to remove the leading option part of the argument (e.g., '`--file=`').

`--source`
`--source=`
`-Wsource=`
The source text is echoed into '`/tmp/ig.s.$$`'.

`--version`
`-Wversion`
`igawk` prints its version number, and runs '`gawk --version`' to get the `gawk` version information, and then exits.

If none of '`-f`', '`--file`', '`-Wfile`', '`--source`', or '`-Wsource`', were supplied, then the first non-option argument should be the `awk` program. If there are no command line arguments left, `igawk` prints an error message and exits. Otherwise, the first argument is echoed into '`/tmp/ig.s.$$`'.

In any case, after the arguments have been processed, '`/tmp/ig.s.$$`' contains the complete text of the original `awk` program.

The '`$$`' in `sh` represents the current process ID number. It is often used in shell programs to generate unique temporary file names. This allows

multiple users to run `igawk` without worrying that the temporary file names will clash.

Here's the program:

```
#! /bin/sh

# igawk --- like gawk but do @include processing
# Arnold Robbins, arnold@gnu.org, Public Domain
# July 1993

if [ "$1" = debug ]
then
    set -x
    shift
else
    # cleanup on exit, hangup, interrupt, quit, termination
    trap 'rm -f /tmp/ig.[se].$$' 0 1 2 3 15
fi

while [ $# -ne 0 ] # loop over arguments
do
    case $1 in
    --)     shift; break;;

    -W)     shift
            set -- -W"$@"
            continue;;

    -[vF])  opts="$opts $1 '$2'"
            shift;;

    -[vF]*) opts="$opts '$1'" ;;

    -f)     echo @include "$2" >> /tmp/ig.s.$$
            shift;;

    -f*)    f=`echo "$1" | sed 's/-f//'`
            echo @include "$f" >> /tmp/ig.s.$$ ;;

    -?file=*)    # -Wfile or --file
            f=`echo "$1" | sed 's/-.file=//'`
            echo @include "$f" >> /tmp/ig.s.$$ ;;

    -?file)    # get arg, $2
            echo @include "$2" >> /tmp/ig.s.$$
```

```
                shift;;

        -?source=*)     # -Wsource or --source
                t=`echo "$1" | sed 's/-.source=//'`
                echo "$t" >> /tmp/ig.s.$$ ;;

        -?source)  # get arg, $2
                echo "$2" >> /tmp/ig.s.$$
                shift;;

        -?version)
                echo igawk: version 1.0 1>&2
                gawk --version
                exit 0 ;;

        -[W-]*)     opts="$opts '$1'" ;;

        *)      break;;
        esac
        shift
    done

    if [ ! -s /tmp/ig.s.$$ ]
    then
        if [ -z "$1" ]
        then
             echo igawk: no program! 1>&2
             exit 1
        else
            echo "$1" > /tmp/ig.s.$$
            shift
        fi
    fi

    # at this point, /tmp/ig.s.$$ has the program
```

The `awk` program to process '`@include`' directives reads through the program, one line at a time using `getline` (see Section 5.8 [Explicit Input with `getline`], page 53). The input file names and '`@include`' statements are managed using a stack. As each '`@include`' is encountered, the current file name is "pushed" onto the stack, and the file named in the '`@include`' directive becomes the current file name. As each file is finished, the stack is "popped," and the previous input file becomes the current input file again. The process is started by making the original file the first one on the stack.

The `pathto` function does the work of finding the full path to a file. It simulates `gawk`'s behavior when searching the `AWKPATH` environment variable (see Section 14.3 [The `AWKPATH` Environment Variable], page 164). If a file name has a '`/`' in it, no path search is done. Otherwise, the file name is concatenated with the name of each directory in the path, and an attempt is made to open the generated file name. The only way in `awk` to test if a file can be read is to go ahead and try to read it with `getline`; that is what `pathto` does.[5] If the file can be read, it is closed, and the file name is returned.

```
gawk -- '
# process @include directives

function pathto(file,    i, t, junk)
{
    if (index(file, "/") != 0)
        return file

    for (i = 1; i <= ndirs; i++) {
        t = (pathlist[i] "/" file)
        if ((getline junk < t) > 0) {
            # found it
            close(t)
            return t
        }
    }
    return ""
}
```

The main program is contained inside one `BEGIN` rule. The first thing it does is set up the `pathlist` array that `pathto` uses. After splitting the path on '`:`', null elements are replaced with `"."`, which represents the current directory.

```
BEGIN {
    path = ENVIRON["AWKPATH"]
    ndirs = split(path, pathlist, ":")
    for (i = 1; i <= ndirs; i++) {
        if (pathlist[i] == "")
            pathlist[i] = "."
    }
```

The stack is initialized with `ARGV[1]`, which will be '`/tmp/ig.s.$$`'. The main loop comes next. Input lines are read in succession. Lines that do not start with '`@include`' are printed verbatim.

---

[5] On some very old versions of `awk`, the test '`getline junk < t`' can loop forever if the file exists but is empty. Caveat Emptor.

If the line does start with '`@include`', the file name is in `$2`. `pathto` is called to generate the full path. If it could not, then we print an error message and continue.

The next thing to check is if the file has been included already. The `processed` array is indexed by the full file name of each included file, and it tracks this information for us. If the file has been seen, a warning message is printed. Otherwise, the new file name is pushed onto the stack and processing continues.

Finally, when `getline` encounters the end of the input file, the file is closed and the stack is popped. When `stackptr` is less than zero, the program is done.

```
        stackptr = 0
        input[stackptr] = ARGV[1] # ARGV[1] is first file

        for (; stackptr >= 0; stackptr--) {
            while ((getline < input[stackptr]) > 0) {
                if (tolower($1) != "@include") {
                    print
                    continue
                }
                fpath = pathto($2)
                if (fpath == "") {
                    printf("igawk:%s:%d: cannot find %s\n", \
                        input[stackptr], FNR, $2) > "/dev/stderr"
                    continue
                }
                if (! (fpath in processed)) {
                    processed[fpath] = input[stackptr]
                    input[++stackptr] = fpath
                } else
                    print $2, "included in", input[stackptr], \
                        "already included in", \
                        processed[fpath] > "/dev/stderr"
            }
            close(input[stackptr])
        }
    }' /tmp/ig.s.$$ > /tmp/ig.e.$$
```

The last step is to call `gawk` with the expanded program and the original options and command line arguments that the user supplied. `gawk`'s exit status is passed back on to `igawk`'s calling program.

```
    eval gawk -f /tmp/ig.e.$$ $opts -- "$@"

    exit $?
```

This version of igawk represents my third attempt at this program. There are three key simplifications that made the program work better.

1. Using '@include' even for the files named with '-f' makes building the initial collected awk program much simpler; all the '@include' processing can be done once.
2. The pathto function doesn't try to save the line read with getline when testing for the file's accessibility. Trying to save this line for use with the main program complicates things considerably.
3. Using a getline loop in the BEGIN rule does it all in one place. It is not necessary to call out to a separate loop for processing nested '@include' statements.

Also, this program illustrates that it is often worthwhile to combine sh and awk programming together. You can usually accomplish quite a lot, without having to resort to low-level programming in C or C++, and it is frequently easier to do certain kinds of string and argument manipulation using the shell than it is in awk.

Finally, igawk shows that it is not always necessary to add new features to a program; they can often be layered on top. With igawk, there is no real reason to build '@include' processing into gawk itself.

As an additional example of this, consider the idea of having two files in a directory in the search path.

'default.awk'
: This file would contain a set of default library functions, such as getopt and assert.

'site.awk'
: This file would contain library functions that are specific to a site or installation, i.e. locally developed functions. Having a separate file allows 'default.awk' to change with new gawk releases, without requiring the system administrator to update it each time by adding the local functions.

One user suggested that gawk be modified to automatically read these files upon startup. Instead, it would be very simple to modify igawk to do this. Since igawk can process nested '@include' directives, 'default.awk' could simply contain '@include' statements for the desired library functions.

17 The Evolution of the awk Language

This book describes the GNU implementation of **awk**, which follows the POSIX specification. Many **awk** users are only familiar with the original **awk** implementation in Version 7 Unix. (This implementation was the basis for **awk** in Berkeley Unix, through 4.3–Reno. The 4.4 release of Berkeley Unix uses **gawk** 2.15.2 for its version of **awk**.) This chapter briefly describes the evolution of the **awk** language, with cross references to other parts of the book where you can find more information.

17.1 Major Changes between V7 and SVR3.1

The **awk** language evolved considerably between the release of Version 7 Unix (1978) and the new version first made generally available in System V Release 3.1 (1987). This section summarizes the changes, with cross-references to further details.

- The requirement for '**;**' to separate rules on a line (see Section 2.6 [**awk** Statements Versus Lines], page 16).
- User-defined functions, and the **return** statement (see Chapter 13 [User-defined Functions], page 151).
- The **delete** statement (see Section 11.6 [The **delete** Statement], page 125).
- The **do-while** statement (see Section 9.3 [The **do-while** Statement], page 104).
- The built-in functions **atan2**, **cos**, **sin**, **rand** and **srand** (see Section 12.2 [Numeric Built-in Functions], page 134).
- The built-in functions **gsub**, **sub**, and **match** (see Section 12.3 [Built-in Functions for String Manipulation], page 135).
- The built-in functions **close**, and **system** (see Section 12.4 [Built-in Functions for Input/Output], page 143).
- The **ARGC**, **ARGV**, **FNR**, **RLENGTH**, **RSTART**, and **SUBSEP** built-in variables (see Chapter 10 [Built-in Variables], page 113).
- The conditional expression using the ternary operator '**?:**' (see Section 7.12 [Conditional Expressions], page 90).
- The exponentiation operator '**^**' (see Section 7.5 [Arithmetic Operators], page 80) and its assignment operator form '**^=**' (see Section 7.7 [Assignment Expressions], page 82).
- C-compatible operator precedence, which breaks some old **awk** programs (see Section 7.14 [Operator Precedence (How Operators Nest)], page 92).
- Regexps as the value of **FS** (see Section 5.5 [Specifying How Fields are Separated], page 44), and as the third argument to the **split** function (see Section 12.3 [Built-in Functions for String Manipulation], page 135).

- Dynamic regexps as operands of the '`~`' and '`!~`' operators (see Section 4.1 [How to Use Regular Expressions], page 23).
- The escape sequences '`\b`', '`\f`', and '`\r`' (see Section 4.2 [Escape Sequences], page 24). (Some vendors have updated their old versions of `awk` to recognize '`\r`', '`\b`', and '`\f`', but this is not something you can rely on.)
- Redirection of input for the `getline` function (see Section 5.8 [Explicit Input with `getline`], page 53).
- Multiple `BEGIN` and `END` rules (see Section 8.1.5 [The `BEGIN` and `END` Special Patterns], page 98).
- Multi-dimensional arrays (see Section 11.9 [Multi-dimensional Arrays], page 128).

17.2 Changes between SVR3.1 and SVR4

The System V Release 4 version of Unix `awk` added these features (some of which originated in `gawk`):

- The `ENVIRON` variable (see Chapter 10 [Built-in Variables], page 113).
- Multiple '`-f`' options on the command line (see Section 14.1 [Command Line Options], page 159).
- The '`-v`' option for assigning variables before program execution begins (see Section 14.1 [Command Line Options], page 159).
- The '`--`' option for terminating command line options.
- The '`\a`', '`\v`', and '`\x`' escape sequences (see Section 4.2 [Escape Sequences], page 24).
- A defined return value for the `srand` built-in function (see Section 12.2 [Numeric Built-in Functions], page 134).
- The `toupper` and `tolower` built-in string functions for case translation (see Section 12.3 [Built-in Functions for String Manipulation], page 135).
- A cleaner specification for the '`%c`' format-control letter in the `printf` function (see Section 6.5.2 [Format-Control Letters], page 63).
- The ability to dynamically pass the field width and precision (`"%*.*d"`) in the argument list of the `printf` function (see Section 6.5.2 [Format-Control Letters], page 63).
- The use of regexp constants such as `/foo/` as expressions, where they are equivalent to using the matching operator, as in '`$0 ~ /foo/`' (see Section 7.2 [Using Regular Expression Constants], page 76).

17.3 Changes between SVR4 and POSIX `awk`

The POSIX Command Language and Utilities standard for `awk` introduced the following changes into the language:

- The use of '-W' for implementation-specific options.
- The use of CONVFMT for controlling the conversion of numbers to strings (see Section 7.4 [Conversion of Strings and Numbers], page 79).
- The concept of a numeric string, and tighter comparison rules to go with it (see Section 7.10 [Variable Typing and Comparison Expressions], page 86).
- More complete documentation of many of the previously undocumented features of the language.

The following common extensions are not permitted by the POSIX standard:

- \x escape sequences are not recognized (see Section 4.2 [Escape Sequences], page 24).
- Newlines do not act as whitespace to separate fields when FS is equal to a single space.
- The synonym func for the keyword function is not recognized (see Section 13.1 [Function Definition Syntax], page 151).
- The operators '**' and '**=' cannot be used in place of '^' and '^=' (see Section 7.5 [Arithmetic Operators], page 80, and also see Section 7.7 [Assignment Expressions], page 82).
- Specifying '-Ft' on the command line does not set the value of FS to be a single tab character (see Section 5.5 [Specifying How Fields are Separated], page 44).
- The fflush built-in function is not supported (see Section 12.4 [Built-in Functions for Input/Output], page 143).

17.4 Extensions in the Bell Laboratories awk

Brian Kernighan, one of the original designers of Unix awk, has made his version available via anonymous ftp (see Section B.8 [Other Freely Available awk Implementations], page 287). This section describes extensions in his version of awk that are not in POSIX awk.

- The '-mf *NNN*' and '-mr *NNN*' command line options to set the maximum number of fields, and the maximum record size, respectively (see Section 14.1 [Command Line Options], page 159).
- The fflush built-in function for flushing buffered output (see Section 12.4 [Built-in Functions for Input/Output], page 143).

17.5 Extensions in gawk Not in POSIX awk

The GNU implementation, gawk, adds a number of features. This sections lists them in the order they were added to gawk. They can all be disabled with either the '--traditional' or '--posix' options (see Section 14.1 [Command Line Options], page 159).

Version 2.10 of `gawk` introduced these features:

- The `AWKPATH` environment variable for specifying a path search for the '`-f`' command line option (see Section 14.1 [Command Line Options], page 159).
- The `IGNORECASE` variable and its effects (see Section 4.5 [Case-sensitivity in Matching], page 33).
- The '`/dev/stdin`', '`/dev/stdout`', '`/dev/stderr`', and '`/dev/fd/`*n*' file name interpretation (see Section 6.7 [Special File Names in `gawk`], page 69).

Version 2.13 of `gawk` introduced these features:

- The `FIELDWIDTHS` variable and its effects (see Section 5.6 [Reading Fixed-width Data], page 49).
- The `systime` and `strftime` built-in functions for obtaining and printing time stamps (see Section 12.5 [Functions for Dealing with Time Stamps], page 146).
- The '`-W lint`' option to provide source code and run time error and portability checking (see Section 14.1 [Command Line Options], page 159).
- The '`-W compat`' option to turn off these extensions (see Section 14.1 [Command Line Options], page 159).
- The '`-W posix`' option for full POSIX compliance (see Section 14.1 [Command Line Options], page 159).

Version 2.14 of `gawk` introduced these features:

- The `next file` statement for skipping to the next data file (see Section 9.8 [The `nextfile` Statement], page 109).

Version 2.15 of `gawk` introduced these features:

- The `ARGIND` variable, that tracks the movement of `FILENAME` through `ARGV` (see Chapter 10 [Built-in Variables], page 113).
- The `ERRNO` variable, that contains the system error message when `getline` returns -1, or when `close` fails (see Chapter 10 [Built-in Variables], page 113).
- The ability to use GNU-style long named options that start with '`--`' (see Section 14.1 [Command Line Options], page 159).
- The '`--source`' option for mixing command line and library file source code (see Section 14.1 [Command Line Options], page 159).
- The '`/dev/pid`', '`/dev/ppid`', '`/dev/pgrpid`', and '`/dev/user`' file name interpretation (see Section 6.7 [Special File Names in `gawk`], page 69).

Version 3.0 of `gawk` introduced these features:

- The `next file` statement became `nextfile` (see Section 9.8 [The `nextfile` Statement], page 109).

- The '`--lint-old`' option to warn about constructs that are not available in the original Version 7 Unix version of `awk` (see Section 17.1 [Major Changes between V7 and SVR3.1], page 247).
- The '`--traditional`' option was added as a better name for '`--compat`' (see Section 14.1 [Command Line Options], page 159).
- The ability for `FS` to be a null string, and for the third argument to `split` to be the null string (see Section 5.5.3 [Making Each Character a Separate Field], page 46).
- The ability for `RS` to be a regexp (see Section 5.1 [How Input is Split into Records], page 37).
- The `RT` variable (see Section 5.1 [How Input is Split into Records], page 37).
- The `gensub` function for more powerful text manipulation (see Section 12.3 [Built-in Functions for String Manipulation], page 135).
- The `strftime` function acquired a default time format, allowing it to be called with no arguments (see Section 12.5 [Functions for Dealing with Time Stamps], page 146).
- Full support for both POSIX and GNU regexps (see Chapter 4 [Regular Expressions], page 23).
- The '`--re-interval`' option to provide interval expressions in regexps (see Section 4.3 [Regular Expression Operators], page 26).
- `IGNORECASE` changed, now applying to string comparison as well as regexp operations (see Section 4.5 [Case-sensitivity in Matching], page 33).
- The '`-m`' option and the `fflush` function from the Bell Labs research version of `awk` (see Section 14.1 [Command Line Options], page 159; also see Section 12.4 [Built-in Functions for Input/Output], page 143).
- The use of GNU Autoconf to control the configuration process (see Section B.2.1 [Compiling `gawk` for Unix], page 278).
- Amiga support (see Section B.6 [Installing `gawk` on an Amiga], page 285).

Appendix A gawk Summary

This appendix provides a brief summary of the `gawk` command line and the `awk` language. It is designed to serve as "quick reference." It is therefore terse, but complete.

A.1 Command Line Options Summary

The command line consists of options to `gawk` itself, the `awk` program text (if not supplied via the '`-f`' option), and values to be made available in the `ARGC` and `ARGV` predefined `awk` variables:

`gawk` [*POSIX or GNU style options*] `-f` *source-file* [`--`] *file* ...
`gawk` [*POSIX or GNU style options*] [`--`] *'program'* *file* ...

The options that `gawk` accepts are:

`-F` *fs*
`--field-separator` *fs*
: Use *fs* for the input field separator (the value of the `FS` predefined variable).

`-f` *program-file*
`--file` *program-file*
: Read the `awk` program source from the file *program-file*, instead of from the first command line argument.

`-mf` *NNN*
`-mr` *NNN*
: The '`f`' flag sets the maximum number of fields, and the '`r`' flag sets the maximum record size. These options are ignored by `gawk`, since `gawk` has no predefined limits; they are only for compatibility with the Bell Labs research version of Unix `awk`.

`-v` *var*=*val*
`--assign` *var*=*val*
: Assign the variable *var* the value *val* before program execution begins.

`-W traditional`
`-W compat`
`--traditional`
`--compat`
: Use compatibility mode, in which `gawk` extensions are turned off.

`-W copyleft`
`-W copyright`
`--copyleft`
`--copyright`
: Print the short version of the General Public License on the standard output, and exit. This option may disappear in a future version of `gawk`.

`-W help`
`-W usage`
`--help`
`--usage` Print a relatively short summary of the available options on the standard output, and exit.

`-W lint`
`--lint` Give warnings about dubious or non-portable `awk` constructs.

`-W lint-old`
`--lint-old`
Warn about constructs that are not available in the original Version 7 Unix version of `awk`.

`-W posix`
`--posix` Use POSIX compatibility mode, in which `gawk` extensions are turned off and additional restrictions apply.

`-W re-interval`
`--re-interval`
Allow interval expressions (see Section 4.3 [Regular Expression Operators], page 26), in regexps.

`-W source=`*program-text*
`--source` *program-text*
Use *program-text* as `awk` program source code. This option allows mixing command line source code with source code from files, and is particularly useful for mixing command line programs with library functions.

`-W version`
`--version`
Print version information for this particular copy of `gawk` on the error output.

`--` Signal the end of options. This is useful to allow further arguments to the `awk` program itself to start with a '`-`'. This is mainly for consistency with POSIX argument parsing conventions.

Any other options are flagged as invalid, but are otherwise ignored. See Section 14.1 [Command Line Options], page 159, for more details.

A.2 Language Summary

An `awk` program consists of a sequence of zero or more pattern-action statements and optional function definitions. One or the other of the pattern and action may be omitted.

pattern `{` *action statements* `}`
pattern
`{` *action statements* `}`

`function` *name*(*parameter list*) `{` *action statements* `}`

`gawk` first reads the program source from the *program-file*(s), if specified, or from the first non-option argument on the command line. The '`-f`' option may be used multiple times on the command line. `gawk` reads the program text from all the *program-file* files, effectively concatenating them in the order they are specified. This is useful for building libraries of `awk` functions, without having to include them in each new `awk` program that uses them. To use a library function in a file from a program typed in on the command line, specify '`--source '`*program*`'`', and type your program in between the single quotes. See Section 14.1 [Command Line Options], page 159.

The environment variable `AWKPATH` specifies a search path to use when finding source files named with the '`-f`' option. The default path, which is '`.:/usr/local/share/awk`'[1] is used if `AWKPATH` is not set. If a file name given to the '`-f`' option contains a '`/`' character, no path search is performed. See Section 14.3 [The `AWKPATH` Environment Variable], page 164.

`gawk` compiles the program into an internal form, and then proceeds to read each file named in the `ARGV` array. The initial values of `ARGV` come from the command line arguments. If there are no files named on the command line, `gawk` reads the standard input.

If a "file" named on the command line has the form '*var*`=`*val*', it is treated as a variable assignment: the variable *var* is assigned the value *val*. If any of the files have a value that is the null string, that element in the list is skipped.

For each record in the input, `gawk` tests to see if it matches any *pattern* in the `awk` program. For each pattern that the record matches, the associated *action* is executed.

A.3 Variables and Fields

`awk` variables are not declared; they come into existence when they are first used. Their values are either floating-point numbers or strings. `awk` also has one-dimensional arrays; multiple-dimensional arrays may be simulated. There are several predefined variables that `awk` sets as a program runs; these are summarized below.

A.3.1 Fields

As each input line is read, `gawk` splits the line into *fields*, using the value of the `FS` variable as the field separator. If `FS` is a single character, fields are separated by that character. Otherwise, `FS` is expected to be a full regular

[1] The path may use a directory other than '`/usr/local/share/awk`', depending upon how `gawk` was built and installed.

expression. In the special case that `FS` is a single space, fields are separated by runs of spaces, tabs and/or newlines.[2] If `FS` is the null string (`""`), then each individual character in the record becomes a separate field. Note that the value of `IGNORECASE` (see Section 4.5 [Case-sensitivity in Matching], page 33) also affects how fields are split when `FS` is a regular expression.

Each field in the input line may be referenced by its position, `$1`, `$2`, and so on. `$0` is the whole line. The value of a field may be assigned to as well. Field numbers need not be constants:

```
n = 5
print $n
```

prints the fifth field in the input line. The variable `NF` is set to the total number of fields in the input line.

References to non-existent fields (i.e. fields after `$NF`) return the null string. However, assigning to a non-existent field (e.g., `$(NF+2) = 5`) increases the value of `NF`, creates any intervening fields with the null string as their value, and causes the value of `$0` to be recomputed, with the fields being separated by the value of `OFS`. Decrementing `NF` causes the values of fields past the new value to be lost, and the value of `$0` to be recomputed, with the fields being separated by the value of `OFS`. See Chapter 5 [Reading Input Files], page 37.

A.3.2 Built-in Variables

`gawk`'s built-in variables are:

`ARGC` The number of elements in `ARGV`. See below for what is actually included in `ARGV`.

`ARGIND` The index in `ARGV` of the current file being processed. When `gawk` is processing the input data files, it is always true that '`FILENAME == ARGV[ARGIND]`'.

`ARGV` The array of command line arguments. The array is indexed from zero to `ARGC` − 1. Dynamically changing `ARGC` and the contents of `ARGV` can control the files used for data. A null-valued element in `ARGV` is ignored. `ARGV` does not include the options to `awk` or the text of the `awk` program itself.

`CONVFMT` The conversion format to use when converting numbers to strings.

`FIELDWIDTHS`
A space separated list of numbers describing the fixed-width input data.

`ENVIRON` An array of environment variable values. The array is indexed by variable name, each element being the value of that variable.

[2] In POSIX `awk`, newline does not separate fields.

Thus, the environment variable `HOME` is `ENVIRON["HOME"]`. One possible value might be '`/home/arnold`'.

Changing this array does not affect the environment seen by programs which `gawk` spawns via redirection or the `system` function. (This may change in a future version of `gawk`.)

Some operating systems do not have environment variables. The `ENVIRON` array is empty when running on these systems.

`ERRNO` The system error message when an error occurs using `getline` or `close`.

`FILENAME` The name of the current input file. If no files are specified on the command line, the value of `FILENAME` is the null string.

`FNR` The input record number in the current input file.

`FS` The input field separator, a space by default.

`IGNORECASE`
The case-sensitivity flag for string comparisons and regular expression operations. If `IGNORECASE` has a non-zero value, then pattern matching in rules, record separating with `RS`, field splitting with `FS`, regular expression matching with '`~`' and '`!~`', and the `gensub`, `gsub`, `index`, `match`, `split` and `sub` built-in functions all ignore case when doing regular expression operations, and all string comparisons are done ignoring case. The value of `IGNORECASE` does *not* affect array subscripting.

`NF` The number of fields in the current input record.

`NR` The total number of input records seen so far.

`OFMT` The output format for numbers for the `print` statement, `"%.6g"` by default.

`OFS` The output field separator, a space by default.

`ORS` The output record separator, by default a newline.

`RS` The input record separator, by default a newline. If `RS` is set to the null string, then records are separated by blank lines. When `RS` is set to the null string, then the newline character always acts as a field separator, in addition to whatever value `FS` may have. If `RS` is set to a multi-character string, it denotes a regexp; input text matching the regexp separates records.

`RT` The input text that matched the text denoted by `RS`, the record separator.

`RSTART` The index of the first character last matched by `match`; zero if no match.

`RLENGTH` The length of the string last matched by `match`; -1 if no match.

`SUBSEP` The string used to separate multiple subscripts in array elements, by default `"\034"`.

See Chapter 10 [Built-in Variables], page 113, for more information.

A.3.3 Arrays

Arrays are subscripted with an expression between square brackets ('`[`' and '`]`'). Array subscripts are *always* strings; numbers are converted to strings as necessary, following the standard conversion rules (see Section 7.4 [Conversion of Strings and Numbers], page 79).

If you use multiple expressions separated by commas inside the square brackets, then the array subscript is a string consisting of the concatenation of the individual subscript values, converted to strings, separated by the subscript separator (the value of `SUBSEP`).

The special operator `in` may be used in a conditional context to see if an array has an index consisting of a particular value.

```
if (val in array)
        print array[val]
```

If the array has multiple subscripts, use '`(i, j, ...) in` *array*' to test for existence of an element.

The `in` construct may also be used in a `for` loop to iterate over all the elements of an array. See Section 11.5 [Scanning All Elements of an Array], page 124.

You can remove an element from an array using the `delete` statement.

You can clear an entire array using '`delete` *array*'.

See Chapter 11 [Arrays in `awk`], page 121.

A.3.4 Data Types

The value of an `awk` expression is always either a number or a string.

Some contexts (such as arithmetic operators) require numeric values. They convert strings to numbers by interpreting the text of the string as a number. If the string does not look like a number, it converts to zero.

Other contexts (such as concatenation) require string values. They convert numbers to strings by effectively printing them with `sprintf`. See Section 7.4 [Conversion of Strings and Numbers], page 79, for the details.

To force conversion of a string value to a number, simply add zero to it. If the value you start with is already a number, this does not change it.

To force conversion of a numeric value to a string, concatenate it with the null string.

Comparisons are done numerically if both operands are numeric, or if one is numeric and the other is a numeric string. Otherwise one or both operands are converted to strings and a string comparison is performed.

Fields, `getline` input, `FILENAME`, `ARGV` elements, `ENVIRON` elements and the elements of an array created by `split` are the only items that can be numeric strings. String constants, such as `"3.1415927"` are not numeric strings, they are string constants. The full rules for comparisons are described in Section 7.10 [Variable Typing and Comparison Expressions], page 86.

Uninitialized variables have the string value `""` (the null, or empty, string). In contexts where a number is required, this is equivalent to zero.

See Section 7.3 [Variables], page 77, for more information on variable naming and initialization; see Section 7.4 [Conversion of Strings and Numbers], page 79, for more information on how variable values are interpreted.

A.4 Patterns

An `awk` program is mostly composed of rules, each consisting of a pattern followed by an action. The action is enclosed in '`{`' and '`}`'. Either the pattern may be missing, or the action may be missing, but not both. If the pattern is missing, the action is executed for every input record. A missing action is equivalent to '`{ print }`', which prints the entire line.

Comments begin with the '`#`' character, and continue until the end of the line. Blank lines may be used to separate statements. Statements normally end with a newline; however, this is not the case for lines ending in a '`,`', '`{`', '`?`', '`:`', '`&&`', or '`||`'. Lines ending in `do` or `else` also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a '`\`', in which case the newline is ignored.

Multiple statements may be put on one line by separating each one with a '`;`'. This applies to both the statements within the action part of a rule (the usual case), and to the rule statements.

See Section 2.2.5 [Comments in `awk` Programs], page 13, for information on `awk`'s commenting convention; see Section 2.6 [`awk` Statements Versus Lines], page 16, for a description of the line continuation mechanism in `awk`.

A.4.1 Pattern Summary

`awk` patterns may be one of the following:

```
/regular expression/
relational expression
pattern && pattern
pattern || pattern
pattern ? pattern : pattern
(pattern)
! pattern
pattern1, pattern2
BEGIN
END
```

`BEGIN` and `END` are two special kinds of patterns that are not tested against the input. The action parts of all `BEGIN` rules are concatenated as if all the statements had been written in a single `BEGIN` rule. They are executed before any of the input is read. Similarly, all the `END` rules are concatenated, and executed when all the input is exhausted (or when an `exit` statement is executed). `BEGIN` and `END` patterns cannot be combined with other patterns in pattern expressions. `BEGIN` and `END` rules cannot have missing action parts.

For */regular-expression/* patterns, the associated statement is executed for each input record that matches the regular expression. Regular expressions are summarized below.

A *relational expression* may use any of the operators defined below in the section on actions. These generally test whether certain fields match certain regular expressions.

The '`&&`', '`||`', and '`!`' operators are logical "and," logical "or," and logical "not," respectively, as in C. They do short-circuit evaluation, also as in C, and are used for combining more primitive pattern expressions. As in most languages, parentheses may be used to change the order of evaluation.

The '`?:`' operator is like the same operator in C. If the first pattern matches, then the second pattern is matched against the input record; otherwise, the third is matched. Only one of the second and third patterns is matched.

The '*pattern1*`,` *pattern2*' form of a pattern is called a range pattern. It matches all input lines starting with a line that matches *pattern1*, and continuing until a line that matches *pattern2*, inclusive. A range pattern cannot be used as an operand of any of the pattern operators.

See Section 8.1 [Pattern Elements], page 95.

A.4.2 Regular Expressions

Regular expressions are based on POSIX EREs (extended regular expressions). The escape sequences allowed in string constants are also valid in regular expressions (see Section 4.2 [Escape Sequences], page 24). Regexps are composed of characters as follows:

c	matches the character *c* (assuming *c* is none of the characters listed below).
`\`*c*	matches the literal character *c*.
`.`	matches any character, *including* newline. In strict POSIX mode, '`.`' does not match the NUL character, which is a character with all bits equal to zero.
`^`	matches the beginning of a string.
`$`	matches the end of a string.

`[`*abc...*`]` matches any of the characters *abc*... (character list).

`[[:`*class*`:]]`
: matches any character in the character class *class*. Allowable classes are `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`.

`[[.`*symbol*`.]]`
: matches the multi-character collating symbol *symbol*. `gawk` does not currently support collating symbols.

`[[=`*classname*`=]]`
: matches any of the equivalent characters in the current locale named by the equivalence class *classname*. `gawk` does not currently support equivalence classes.

`[^`*abc...*`]`
: matches any character except *abc*... (negated character list).

r1`|`*r2* matches either *r1* or *r2* (alternation).

r1r2 matches *r1*, and then *r2* (concatenation).

r`+` matches one or more *r*'s.

r`*` matches zero or more *r*'s.

r`?` matches zero or one *r*'s.

`(`*r*`)` matches *r* (grouping).

r`{`*n*`}`
r`{`*n*`,}`
r`{`*n*`,`*m*`}` matches at least *n*, *n* to any number, or *n* to *m* occurrences of *r* (interval expressions).

`\y` matches the empty string at either the beginning or the end of a word.

`\B` matches the empty string within a word.

`\<` matches the empty string at the beginning of a word.

`\>` matches the empty string at the end of a word.

`\w` matches any word-constituent character (alphanumeric characters and the underscore).

`\W` matches any character that is not word-constituent.

`\‘` matches the empty string at the beginning of a buffer (same as a string in `gawk`).

`\’` matches the empty string at the end of a buffer.

The various command line options control how `gawk` interprets characters in regexps.

No options
: In the default case, `gawk` provide all the facilities of POSIX regexps and the GNU regexp operators described above. However, interval expressions are not supported.

`--posix`
: Only POSIX regexps are supported, the GNU operators are not special (e.g., '`\w`' matches a literal '`w`'). Interval expressions are allowed.

`--traditional`
: Traditional Unix `awk` regexps are matched. The GNU operators are not special, interval expressions are not available, and neither are the POSIX character classes (`[[:alnum:]]` and so on). Characters described by octal and hexadecimal escape sequences are treated literally, even if they represent regexp metacharacters.

`--re-interval`
: Allow interval expressions in regexps, even if '`--traditional`' has been provided.

See Chapter 4 [Regular Expressions], page 23.

A.5 Actions

Action statements are enclosed in braces, '`{`' and '`}`'. A missing action statement is equivalent to '`{ print }`'.

Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and Input/Output statements available are similar to those in C.

Comments begin with the '`#`' character, and continue until the end of the line. Blank lines may be used to separate statements. Statements normally end with a newline; however, this is not the case for lines ending in a '`,`', '`{`', '`?`', '`:`', '`&&`', or '`||`'. Lines ending in `do` or `else` also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a '`\`', in which case the newline is ignored.

Multiple statements may be put on one line by separating each one with a '`;`'. This applies to both the statements within the action part of a rule (the usual case), and to the rule statements.

See Section 2.2.5 [Comments in `awk` Programs], page 13, for information on `awk`'s commenting convention; see Section 2.6 [`awk` Statements Versus Lines], page 16, for a description of the line continuation mechanism in `awk`.

A.5.1 Operators

The operators in `awk`, in order of decreasing precedence, are:

`(...)`
: Grouping.

`$` Field reference.

`++ --` Increment and decrement, both prefix and postfix.

`^` Exponentiation ('`**`' may also be used, and '`**=`' for the assignment operator, but they are not specified in the POSIX standard).

`+ - !` Unary plus, unary minus, and logical negation.

`* / %` Multiplication, division, and modulus.

`+ -` Addition and subtraction.

space String concatenation.

`< <= > >= != ==`
The usual relational operators.

`~ !~` Regular expression match, negated match.

`in` Array membership.

`&&` Logical "and".

`||` Logical "or".

`?:` A conditional expression. This has the form '*expr1* `?` *expr2* `:` *expr3*'. If *expr1* is true, the value of the expression is *expr2*; otherwise it is *expr3*. Only one of *expr2* and *expr3* is evaluated.

`= += -= *= /= %= ^=`
Assignment. Both absolute assignment (*var*`=`*value*) and operator assignment (the other forms) are supported.

See Chapter 7 [Expressions], page 75.

A.5.2 Control Statements

The control statements are as follows:

```
if (condition) statement [ else statement ]
while (condition) statement
do statement while (condition)
for (expr1; expr2; expr3) statement
for (var in array) statement
break
continue
delete array[index]
delete array
exit [ expression ]
{ statements }
```

See Chapter 9 [Control Statements in Actions], page 103.

A.5.3 I/O Statements

The Input/Output statements are as follows:

`getline` Set `$0` from next input record; set `NF`, `NR`, `FNR`. See Section 5.8 [Explicit Input with `getline`], page 53.

`getline <`*file*
Set `$0` from next record of *file*; set `NF`.

`getline` *var*
Set *var* from next input record; set `NR`, `FNR`.

`getline` *var* `<`*file*
Set *var* from next record of *file*.

command `| getline`
Run *command*, piping its output into `getline`; sets `$0`, `NF`, `NR`.

command `| getline var`
Run *command*, piping its output into `getline`; sets *var*.

`next` Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the `awk` program. If the end of the input data is reached, the `END` rule(s), if any, are executed. See Section 9.7 [The `next` Statement], page 108.

`nextfile` Stop processing the current input file. The next input record read comes from the next input file. `FILENAME` is updated, `FNR` is set to one, `ARGIND` is incremented, and processing starts over with the first pattern in the `awk` program. If the end of the input data is reached, the `END` rule(s), if any, are executed. Earlier versions of `gawk` used '`next file`'; this usage is still supported, but is considered to be deprecated. See Section 9.8 [The `nextfile` Statement], page 109.

`print` Prints the current record. See Chapter 6 [Printing Output], page 59.

`print` *expr-list*
Prints expressions.

`print` *expr-list* `>` *file*
Prints expressions to *file*. If *file* does not exist, it is created. If it does exist, its contents are deleted the first time the `print` is executed.

`print` *expr-list* `>>` *file*
Prints expressions to *file*. The previous contents of *file* are retained, and the output of `print` is appended to the file.

`print` *expr-list* `|` *command*
: Prints expressions, sending the output down a pipe to *command*. The pipeline to the command stays open until the `close` function is called.

`printf` *fmt*`,` *expr-list*
: Format and print.

`printf` *fmt*`,` *expr-list* `>` *file*
: Format and print to *file*. If *file* does not exist, it is created. If it does exist, its contents are deleted the first time the `printf` is executed.

`printf` *fmt*`,` *expr-list* `>>` *file*
: Format and print to *file*. The previous contents of *file* are retained, and the output of `printf` is appended to the file.

`printf` *fmt*`,` *expr-list* `|` *command*
: Format and print, sending the output down a pipe to *command*. The pipeline to the command stays open until the `close` function is called.

`getline` returns zero on end of file, and -1 on an error. In the event of an error, `getline` will set `ERRNO` to the value of a system-dependent string that describes the error.

A.5.4 `printf` Summary

Conversion specification have the form `%`[*flag*][*width*][`.`*prec*]*format*. Items in brackets are optional.

The `awk printf` statement and `sprintf` function accept the following conversion specification formats:

`%c`
: An ASCII character. If the argument used for '`%c`' is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

`%d`
`%i`
: A decimal number (the integer part).

`%e`
`%E`
: A floating point number of the form '[`-`]`d.dddddde`[`+-`]`dd`'. The '`%E`' format uses '`E`' instead of '`e`'.

`%f`
: A floating point number of the form [`-`]`ddd.dddddd`.

`%g`
`%G`
: Use either the '`%e`' or '`%f`' formats, whichever produces a shorter string, with non-significant zeros suppressed. '`%G`' will use '`%E`' instead of '`%e`'.

`%o` An unsigned octal number (also an integer).

`%u` An unsigned decimal number (again, an integer).

`%s` A character string.

`%x`
`%X` An unsigned hexadecimal number (an integer). The '`%X`' format uses '`A`' through '`F`' instead of '`a`' through '`f`' for decimal 10 through 15.

`%%` A single '`%`' character; no argument is converted.

There are optional, additional parameters that may lie between the '`%`' and the control letter:

`-` The expression should be left-justified within its field.

space For numeric conversions, prefix positive values with a space, and negative values with a minus sign.

`+` The plus sign, used before the width modifier (see below), says to always supply a sign for numeric conversions, even if the data to be formatted is positive. The '`+`' overrides the space modifier.

`#` Use an "alternate form" for certain control letters. For '`o`', supply a leading zero. For '`x`', and '`X`', supply a leading '`0x`' or '`0X`' for a non-zero result. For '`e`', '`E`', and '`f`', the result will always contain a decimal point. For '`g`', and '`G`', trailing zeros are not removed from the result.

`0` A leading '`0`' (zero) acts as a flag, that indicates output should be padded with zeros instead of spaces. This applies even to non-numeric output formats. This flag only has an effect when the field width is wider than the value to be printed.

width The field should be padded to this width. The field is normally padded with spaces. If the '`0`' flag has been used, it is padded with zeros.

.prec A number that specifies the precision to use when printing. For the '`e`', '`E`', and '`f`' formats, this specifies the number of digits you want printed to the right of the decimal point. For the '`g`', and '`G`' formats, it specifies the maximum number of significant digits. For the '`d`', '`o`', '`i`', '`u`', '`x`', and '`X`' formats, it specifies the minimum number of digits to print. For the '`s`' format, it specifies the maximum number of characters from the string that should be printed.

Either or both of the *width* and *prec* values may be specified as '`*`'. In that case, the particular value is taken from the argument list.

See Section 6.5 [Using `printf` Statements for Fancier Printing], page 62.

A.5.5 Special File Names

When doing I/O redirection from either `print` or `printf` into a file, or via `getline` from a file, `gawk` recognizes certain special file names internally. These file names allow access to open file descriptors inherited from `gawk`'s parent process (usually the shell). The file names are:

'`/dev/stdin`'
: The standard input.

'`/dev/stdout`'
: The standard output.

'`/dev/stderr`'
: The standard error output.

'`/dev/fd/`*n*'
: The file denoted by the open file descriptor *n*.

In addition, reading the following files provides process related information about the running `gawk` program. All returned records are terminated with a newline.

'`/dev/pid`'
: Returns the process ID of the current process.

'`/dev/ppid`'
: Returns the parent process ID of the current process.

'`/dev/pgrpid`'
: Returns the process group ID of the current process.

'`/dev/user`'
: At least four space-separated fields, containing the return values of the `getuid`, `geteuid`, `getgid`, and `getegid` system calls. If there are any additional fields, they are the group IDs returned by `getgroups` system call. (Multiple groups may not be supported on all systems.)

These file names may also be used on the command line to name data files. These file names are only recognized internally if you do not actually have files with these names on your system.

See Section 6.7 [Special File Names in `gawk`], page 69, for a longer description that provides the motivation for this feature.

A.5.6 Built-in Functions

`awk` provides a number of built-in functions for performing numeric operations, string related operations, and I/O related operations.

The built-in arithmetic functions are:

`atan2(`*y*`, `*x*`)`
: the arctangent of y/x in radians.

`cos(`*expr*`)`
: the cosine of *expr*, which is in radians.

`exp(`*expr*`)`
: the exponential function (`e ^` *expr*).

`int(`*expr*`)`
: truncates to integer.

`log(`*expr*`)`
: the natural logarithm of `expr`.

`rand()`
: a random number between zero and one.

`sin(`*expr*`)`
: the sine of *expr*, which is in radians.

`sqrt(`*expr*`)`
: the square root function.

`srand(`[*expr*]`)`
: use *expr* as a new seed for the random number generator. If no *expr* is provided, the time of day is used. The return value is the previous seed for the random number generator.

`awk` has the following built-in string functions:

`gensub(`*regex*`, `*subst*`, `*how* [`, `*target*]`)`
: If *how* is a string beginning with '`g`' or '`G`', then replace each match of *regex* in *target* with *subst*. Otherwise, replace the *how*'th occurrence. If *target* is not supplied, use `$0`. The return value is the changed string; the original *target* is not modified. Within *subst*, '`\`*n*', where *n* is a digit from one to nine, can be used to indicate the text that matched the *n*'th parenthesized subexpression. This function is `gawk`-specific.

`gsub(`*regex*`, `*subst* [`, `*target*]`)`
: for each substring matching the regular expression *regex* in the string *target*, substitute the string *subst*, and return the number of substitutions. If *target* is not supplied, use `$0`.

`index(`*str*`, `*search*`)`
: returns the index of the string *search* in the string *str*, or zero if *search* is not present.

`length(`[*str*]`)`
: returns the length of the string *str*. The length of `$0` is returned if no argument is supplied.

`match(`*str*`, `*regex*`)`
: returns the position in *str* where the regular expression *regex* occurs, or zero if *regex* is not present, and sets the values of `RSTART` and `RLENGTH`.

`split(`*str*`, `*arr* `[, `*regex*`])`
: splits the string *str* into the array *arr* on the regular expression *regex*, and returns the number of elements. If *regex* is omitted, `FS` is used instead. *regex* can be the null string, causing each character to be placed into its own array element. The array *arr* is cleared first.

`sprintf(`*fmt*`, `*expr-list*`)`
: prints *expr-list* according to *fmt*, and returns the resulting string.

`sub(`*regex*`, `*subst* `[, `*target*`])`
: just like `gsub`, but only the first matching substring is replaced.

`substr(`*str*`, `*index* `[, `*len*`])`
: returns the *len*-character substring of *str* starting at *index*. If *len* is omitted, the rest of *str* is used.

`tolower(`*str*`)`
: returns a copy of the string *str*, with all the upper-case characters in *str* translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

`toupper(`*str*`)`
: returns a copy of the string *str*, with all the lower-case characters in *str* translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

The I/O related functions are:

`close(`*expr*`)`
: Close the open file or pipe denoted by *expr*.

`fflush(`[*expr*]`)`
: Flush any buffered output for the output file or pipe denoted by *expr*. If *expr* is omitted, standard output is flushed. If *expr* is the null string (`""`), all output buffers are flushed.

`system(`*cmd-line*`)`
: Execute the command *cmd-line*, and return the exit status. If your operating system does not support `system`, calling it will generate a fatal error.

  '`system("")`' can be used to force `awk` to flush any pending output. This is more portable, but less obvious, than calling `fflush`.

A.5.7 Time Functions

The following two functions are available for getting the current time of day, and for formatting time stamps. They are specific to `gawk`.

`systime()`
: returns the current time of day as the number of seconds since a particular epoch (Midnight, January 1, 1970 UTC, on POSIX systems).

`strftime(`[*format*[`,` *timestamp*]]`)`
: formats *timestamp* according to the specification in *format*. The current time of day is used if no *timestamp* is supplied. A default format equivalent to the output of the `date` utility is used if no *format* is supplied. See Section 12.5 [Functions for Dealing with Time Stamps], page 146, for the details on the conversion specifiers that `strftime` accepts.

See Chapter 12 [Built-in Functions], page 133, for a description of all of `awk`'s built-in functions.

A.5.8 String Constants

String constants in `awk` are sequences of characters enclosed in double quotes (`"`). Within strings, certain *escape sequences* are recognized, as in C. These are:

`\\`
: A literal backslash.

`\a`
: The "alert" character; usually the ASCII BEL character.

`\b`
: Backspace.

`\f`
: Formfeed.

`\n`
: Newline.

`\r`
: Carriage return.

`\t`
: Horizontal tab.

`\v`
: Vertical tab.

`\x`*hex digits*
: The character represented by the string of hexadecimal digits following the '`\x`'. As in ANSI C, all following hexadecimal digits are considered part of the escape sequence. E.g., `"\x1B"` is a string containing the ASCII ESC (escape) character. (The '`\x`' escape sequence is not in POSIX `awk`.)

`\`*ddd*
: The character represented by the one, two, or three digit sequence of octal digits. Thus, `"\033"` is also a string containing the ASCII ESC (escape) character.

`\`*c*
: The literal character *c*, if *c* is not one of the above.

The escape sequences may also be used inside constant regular expressions (e.g., the regexp `/[ \t\f\n\r\v]/` matches whitespace characters).

See Section 4.2 [Escape Sequences], page 24.

A.6 User-defined Functions

Functions in `awk` are defined as follows:

`function` *name*`(`*parameter list*`) { `*statements*` }`

Actual parameters supplied in the function call are used to instantiate the formal parameters declared in the function. Arrays are passed by reference, other variables are passed by value.

If there are fewer arguments passed than there are names in *parameter-list*, the extra names are given the null string as their value. Extra names have the effect of local variables.

The open-parenthesis in a function call of a user-defined function must immediately follow the function name, without any intervening white space. This is to avoid a syntactic ambiguity with the concatenation operator.

The word `func` may be used in place of `function` (but not in POSIX `awk`).

Use the `return` statement to return a value from a function.

See Chapter 13 [User-defined Functions], page 151.

A.7 Historical Features

There are two features of historical `awk` implementations that `gawk` supports.

First, it is possible to call the `length` built-in function not only with no arguments, but even without parentheses!

```
a = length
```

is the same as either of

```
a = length()
a = length($0)
```

For example:

```
$ echo abcdef | awk '{ print length }'
⊣ 6
```

This feature is marked as "deprecated" in the POSIX standard, and `gawk` will issue a warning about its use if '`--lint`' is specified on the command line. (The ability to use `length` this way was actually an accident of the original Unix `awk` implementation. If any built-in function used `$0` as its default argument, it was possible to call that function without the parentheses. In particular, it was common practice to use the `length` function in this fashion, and this usage was documented in the `awk` manual page.)

The other historical feature is the use of either the `break` statement, or the `continue` statement outside the body of a `while`, `for`, or `do` loop. Traditional `awk` implementations have treated such usage as equivalent to the `next` statement. More recent versions of Unix `awk` do not allow it. `gawk` supports this usage if '`--traditional`' has been specified.

See Section 14.1 [Command Line Options], page 159, for more information about the '`--posix`' and '`--lint`' options.

Appendix B Installing gawk

This appendix provides instructions for installing `gawk` on the various platforms that are supported by the developers. The primary developers support Unix (and one day, GNU), while the other ports were contributed. The file '`ACKNOWLEDGMENT`' in the `gawk` distribution lists the electronic mail addresses of the people who did the respective ports, and they are also provided in Section B.7 [Reporting Problems and Bugs], page 285.

B.1 The gawk Distribution

This section first describes how to get the `gawk` distribution, how to extract it, and then what is in the various files and subdirectories.

B.1.1 Getting the gawk Distribution

There are three ways you can get GNU software.

1. You can copy it from someone else who already has it.
2. You can order `gawk` directly from the Free Software Foundation. Software distributions are available for Unix, MS-DOS, and VMS, on tape and CD-ROM. The address is:

   Free Software Foundation
   59 Temple Place—Suite 330
   Boston, MA 02111-1307 USA
   Phone: +1-617-542-5942
   Fax (including Japan): +1-617-542-2652
   Email: `gnu@gnu.org`
   URL: `http://www.gnu.org/`

   Ordering from the FSF directly contributes to the support of the foundation and to the production of more free software.
3. You can get `gawk` by using anonymous `ftp` to the Internet host `gnudist.gnu.org`, in the directory '`/gnu/gawk`'.

   Here is a list of alternate `ftp` sites from which you can obtain GNU software. When a site is listed as "*site*`:`*directory*" the *directory* indicates the directory where GNU software is kept. You should use a site that is geographically close to you.

   Asia:

   ```
   cair-archive.kaist.ac.kr:/pub/gnu
   ftp.cs.titech.ac.jp
   ftp.nectec.or.th:/pub/mirrors/gnu
   utsun.s.u-tokyo.ac.jp:/ftpsync/prep
   ```

Australia:

`archie.au:/gnu`
(`archie.oz` or `archie.oz.au` for ACSnet)

Africa:

`ftp.sun.ac.za:/pub/gnu`

Middle East:

`ftp.technion.ac.il:/pub/unsupported/gnu`

Europe:

`archive.eu.net`
`ftp.denet.dk`
`ftp.eunet.ch`
`ftp.funet.fi:/pub/gnu`
`ftp.ieunet.ie:pub/gnu`
`ftp.informatik.rwth-aachen.de:/pub/gnu`
`ftp.informatik.tu-muenchen.de`
`ftp.luth.se:/pub/unix/gnu`
`ftp.mcc.ac.uk`
`ftp.stacken.kth.se`
`ftp.sunet.se:/pub/gnu`
`ftp.univ-lyon1.fr:pub/gnu`
`ftp.win.tue.nl:/pub/gnu`
`irisa.irisa.fr:/pub/gnu`
`isy.liu.se`
`nic.switch.ch:/mirror/gnu`
`src.doc.ic.ac.uk:/gnu`
`unix.hensa.ac.uk:/pub/uunet/systems/gnu`

South America:

`ftp.inf.utfsm.cl:/pub/gnu`
`ftp.unicamp.br:/pub/gnu`

Western Canada:

`ftp.cs.ubc.ca:/mirror2/gnu`

USA:

`col.hp.com:/mirrors/gnu`
`f.ms.uky.edu:/pub3/gnu`
`ftp.cc.gatech.edu:/pub/gnu`
`ftp.cs.columbia.edu:/archives/gnu/prep`
`ftp.digex.net:/pub/gnu`
`ftp.hawaii.edu:/mirrors/gnu`
`ftp.kpc.com:/pub/mirror/gnu`

USA (continued):

```
ftp.uu.net:/systems/gnu
gatekeeper.dec.com:/pub/GNU
jaguar.utah.edu:/gnustuff
labrea.stanford.edu
mrcnext.cso.uiuc.edu:/pub/gnu
vixen.cso.uiuc.edu:/gnu
wuarchive.wustl.edu:/systems/gnu
```

B.1.2 Extracting the Distribution

`gawk` is distributed as a `tar` file compressed with the GNU Zip program, `gzip`.

Once you have the distribution (for example, '`gawk-3.0.6.tar.gz`'), first use `gzip` to expand the file, and then use `tar` to extract it. You can use the following pipeline to produce the `gawk` distribution:

```
# Under System V, add 'o' to the tar flags
gzip -d -c gawk-3.0.6.tar.gz | tar -xvpf -
```

This will create a directory named '`gawk-3.0.6`' in the current directory.

The distribution file name is of the form '`gawk-`*V*`.`*R*`.`*n*`.tar.gz`'. The *V* represents the major version of `gawk`, the *R* represents the current release of version *V*, and the *n* represents a *patch level*, meaning that minor bugs have been fixed in the release. The current patch level is 6, but when retrieving distributions, you should get the version with the highest version, release, and patch level. (Note that release levels greater than or equal to 90 denote "beta," or non-production software; you may not wish to retrieve such a version unless you don't mind experimenting.)

If you are not on a Unix system, you will need to make other arrangements for getting and extracting the `gawk` distribution. You should consult a local expert.

B.1.3 Contents of the `gawk` Distribution

The `gawk` distribution has a number of C source files, documentation files, subdirectories and files related to the configuration process (see Section B.2 [Compiling and Installing `gawk` on Unix], page 278), and several subdirectories related to different, non-Unix, operating systems.

various '`.c`', '`.y`', and '`.h`' files
: These files are the actual `gawk` source code.

'`README`'
'`README_d/README.*`'
: Descriptive files: '`README`' for `gawk` under Unix, and the rest for the various hardware and software combinations.

'`INSTALL`' A file providing an overview of the configuration and installation process.

'`PORTS`' A list of systems to which `gawk` has been ported, and which have successfully run the test suite.

'`ACKNOWLEDGMENT`'
A list of the people who contributed major parts of the code or documentation.

'`ChangeLog`'
A detailed list of source code changes as bugs are fixed or improvements made.

'`NEWS`' A list of changes to `gawk` since the last release or patch.

'`COPYING`' The GNU General Public License.

'`FUTURES`' A brief list of features and/or changes being contemplated for future releases, with some indication of the time frame for the feature, based on its difficulty.

'`LIMITATIONS`'
A list of those factors that limit `gawk`'s performance. Most of these depend on the hardware or operating system software, and are not limits in `gawk` itself.

'`POSIX.STD`'
A description of one area where the POSIX standard for `awk` is incorrect, and how `gawk` handles the problem.

'`PROBLEMS`'
A file describing known problems with the current release.

'`doc/awkforai.txt`'
A short article describing why `gawk` is a good language for AI (Artificial Intelligence) programming.

'`doc/README.card`'
'`doc/ad.block`'
'`doc/awkcard.in`'
'`doc/cardfonts`'
'`doc/colors`'
'`doc/macros`'
'`doc/no.colors`'
'`doc/setter.outline`'
The `troff` source for a five-color `awk` reference card. A modern version of `troff`, such as GNU Troff (`groff`) is needed to produce the color version. See the file '`README.card`' for instructions if you have an older `troff`.

'`doc/gawk.1`'
: The `troff` source for a manual page describing `gawk`. This is distributed for the convenience of Unix users.

'`doc/gawk.texi`'
: The Texinfo source file for this book. It should be processed with TEX to produce a printed document, and with `makeinfo` to produce an Info file.

'`doc/gawk.info`'
: The generated Info file for this book.

'`doc/igawk.1`'
: The `troff` source for a manual page describing the `igawk` program presented in Section 16.2.9 [An Easy Way to Use Library Functions], page 239.

'`doc/Makefile.in`'
: The input file used during the configuration process to generate the actual '`Makefile`' for creating the documentation.

'`Makefile.in`'
'`acconfig.h`'
'`aclocal.m4`'
'`configh.in`'
'`configure.in`'
'`configure`'
'`custom.h`'
'`missing/*`'
: These files and subdirectory are used when configuring `gawk` for various Unix systems. They are explained in detail in Section B.2 [Compiling and Installing `gawk` on Unix], page 278.

'`awklib/extract.awk`'
'`awklib/Makefile.in`'
: The '`awklib`' directory contains a copy of '`extract.awk`' (see Section 16.2.7 [Extracting Programs from Texinfo Source Files], page 233), which can be used to extract the sample programs from the Texinfo source file for this book, and a '`Makefile.in`' file, which `configure` uses to generate a '`Makefile`'. As part of the process of building `gawk`, the library functions from Chapter 15 [A Library of `awk` Functions], page 167, and the `igawk` program from Section 16.2.9 [An Easy Way to Use Library Functions], page 239, are extracted into ready to use files. They are installed as part of the installation process.

'`atari/*`'
: Files needed for building `gawk` on an Atari ST. See Section B.5 [Installing `gawk` on the Atari ST], page 283, for details.

'`pc/*`' Files needed for building `gawk` under MS-DOS and OS/2. See Section B.4 [MS-DOS and OS/2 Installation and Compilation], page 282, for details.

'`vms/*`' Files needed for building `gawk` under VMS. See Section B.3 [How to Compile and Install `gawk` on VMS], page 279, for details.

'`test/*`' A test suite for `gawk`. You can use '`make check`' from the top level `gawk` directory to run your version of `gawk` against the test suite. If `gawk` successfully passes '`make check`' then you can be confident of a successful port.

B.2 Compiling and Installing `gawk` on Unix

Usually, you can compile and install `gawk` by typing only two commands. However, if you do use an unusual system, you may need to configure `gawk` for your system yourself.

B.2.1 Compiling `gawk` for Unix

After you have extracted the `gawk` distribution, `cd` to '`gawk-3.0.6`'. Like most GNU software, `gawk` is configured automatically for your Unix system by running the `configure` program. This program is a Bourne shell script that was generated automatically using GNU `autoconf`. (The `autoconf` software is described fully in *Autoconf—Generating Automatic Configuration Scripts*, which is available from the Free Software Foundation.)

To configure `gawk`, simply run `configure`:

```
sh ./configure
```

This produces a '`Makefile`' and '`config.h`' tailored to your system. The '`config.h`' file describes various facts about your system. You may wish to edit the '`Makefile`' to change the `CFLAGS` variable, which controls the command line options that are passed to the C compiler (such as optimization levels, or compiling for debugging).

Alternatively, you can add your own values for most `make` variables, such as `CC` and `CFLAGS`, on the command line when running `configure`:

```
CC=cc CFLAGS=-g sh ./configure
```

See the file '`INSTALL`' in the `gawk` distribution for all the details.

After you have run `configure`, and possibly edited the '`Makefile`', type:

```
make
```

and shortly thereafter, you should have an executable version of `gawk`. That's all there is to it! (If these steps do not work, please send in a bug report; see Section B.7 [Reporting Problems and Bugs], page 285.)

B.2.2 The Configuration Process

(This section is of interest only if you know something about using the C language and the Unix operating system.)

The source code for `gawk` generally attempts to adhere to formal standards wherever possible. This means that `gawk` uses library routines that are specified by the ANSI C standard and by the POSIX operating system interface standard. When using an ANSI C compiler, function prototypes are used to help improve the compile-time checking.

Many Unix systems do not support all of either the ANSI or the POSIX standards. The '`missing`' subdirectory in the `gawk` distribution contains replacement versions of those subroutines that are most likely to be missing.

The '`config.h`' file that is created by the `configure` program contains definitions that describe features of the particular operating system where you are attempting to compile `gawk`. The three things described by this file are what header files are available, so that they can be correctly included, what (supposedly) standard functions are actually available in your C libraries, and other miscellaneous facts about your variant of Unix. For example, there may not be an `st_blksize` element in the `stat` structure. In this case '`HAVE_ST_BLKSIZE`' would be undefined.

It is possible for your C compiler to lie to `configure`. It may do so by not exiting with an error when a library function is not available. To get around this, you can edit the file '`custom.h`'. Use an '`#ifdef`' that is appropriate for your system, and either `#define` any constants that `configure` should have defined but didn't, or `#undef` any constants that `configure` defined and should not have. '`custom.h`' is automatically included by '`config.h`'.

It is also possible that the `configure` program generated by `autoconf` will not work on your system in some other fashion. If you do have a problem, the file '`configure.in`' is the input for `autoconf`. You may be able to change this file, and generate a new version of `configure` that will work on your system. See Section B.7 [Reporting Problems and Bugs], page 285, for information on how to report problems in configuring `gawk`. The same mechanism may be used to send in updates to '`configure.in`' and/or '`custom.h`'.

B.3 How to Compile and Install gawk on VMS

This section describes how to compile and install `gawk` under VMS.

B.3.1 Compiling gawk on VMS

To compile `gawk` under VMS, there is a `DCL` command procedure that will issue all the necessary `CC` and `LINK` commands, and there is also a '`Makefile`' for use with the `MMS` utility. From the source directory, use either

```
$ @[.VMS]VMSBUILD.COM
```

or

```
$ MMS/DESCRIPTION=[.VMS]DESCRIP.MMS GAWK
```

Depending upon which C compiler you are using, follow one of the sets of instructions in this table:

VAX C V3.x
: Use either '`vmsbuild.com`' or '`descrip.mms`' as is. These use `CC/OPTIMIZE=NOLINE`, which is essential for Version 3.0.

VAX C V2.x
: You must have Version 2.3 or 2.4; older ones won't work. Edit either '`vmsbuild.com`' or '`descrip.mms`' according to the comments in them. For '`vmsbuild.com`', this just entails removing two '`!`' delimiters. Also edit '`config.h`' (which is a copy of file '`[.config]vms-conf.h`') and comment out or delete the two lines '`#define __STDC__ 0`' and '`#define VAXC_BUILTINS`' near the end.

GNU C
: Edit '`vmsbuild.com`' or '`descrip.mms`'; the changes are different from those for VAX C V2.x, but equally straightforward. No changes to '`config.h`' should be needed.

DEC C
: Edit '`vmsbuild.com`' or '`descrip.mms`' according to their comments. No changes to '`config.h`' should be needed.

`gawk` has been tested under VAX/VMS 5.5-1 using VAX C V3.2, GNU C 1.40 and 2.3. It should work without modifications for VMS V4.6 and up.

B.3.2 Installing `gawk` on VMS

To install `gawk`, all you need is a "foreign" command, which is a `DCL` symbol whose value begins with a dollar sign. For example:

```
$ GAWK :== $disk1:[gnubin]GAWK
```

(Substitute the actual location of `gawk.exe` for '`$disk1:[gnubin]`'.) The symbol should be placed in the '`login.com`' of any user who wishes to run `gawk`, so that it will be defined every time the user logs on. Alternatively, the symbol may be placed in the system-wide '`sylogin.com`' procedure, which will allow all users to run `gawk`.

Optionally, the help entry can be loaded into a VMS help library:

```
$ LIBRARY/HELP SYS$HELP:HELPLIB [.VMS]GAWK.HLP
```

(You may want to substitute a site-specific help library rather than the standard VMS library '`HELPLIB`'.) After loading the help text,

```
$ HELP GAWK
```

will provide information about both the `gawk` implementation and the `awk` programming language.

The logical name '`AWK_LIBRARY`' can designate a default location for `awk` program files. For the '`-f`' option, if the specified filename has no device or directory path information in it, `gawk` will look in the current directory

first, then in the directory specified by the translation of 'AWK_LIBRARY' if the file was not found. If after searching in both directories, the file still is not found, then gawk appends the suffix '.awk' to the filename and the file search will be re-tried. If 'AWK_LIBRARY' is not defined, that portion of the file search will fail benignly.

B.3.3 Running gawk on VMS

Command line parsing and quoting conventions are significantly different on VMS, so examples in this book or from other sources often need minor changes. They *are* minor though, and all awk programs should run correctly.

Here are a couple of trivial tests:

```
$ gawk -- "BEGIN {print ""Hello, World!""}"
$ gawk -"W" version
! could also be -"W version" or "-W version"
```

Note that upper-case and mixed-case text must be quoted.

The VMS port of gawk includes a DCL-style interface in addition to the original shell-style interface (see the help entry for details). One side-effect of dual command line parsing is that if there is only a single parameter (as in the quoted string program above), the command becomes ambiguous. To work around this, the normally optional '--' flag is required to force Unix style rather than DCL parsing. If any other dash-type options (or multiple parameters such as data files to be processed) are present, there is no ambiguity and '--' can be omitted.

The default search path when looking for awk program files specified by the '-f' option is "SYS$DISK:[],AWK_LIBRARY:". The logical name 'AWKPATH' can be used to override this default. The format of 'AWKPATH' is a comma-separated list of directory specifications. When defining it, the value should be quoted so that it retains a single translation, and not a multi-translation RMS searchlist.

B.3.4 Building and Using gawk on VMS POSIX

Ignore the instructions above, although 'vms/gawk.hlp' should still be made available in a help library. The source tree should be unpacked into a container file subsystem rather than into the ordinary VMS file system. Make sure that the two scripts, 'configure' and 'vms/posix-cc.sh', are executable; use 'chmod +x' on them if necessary. Then execute the following two commands:

```
psx> CC=vms/posix-cc.sh configure
psx> make CC=c89 gawk
```

The first command will construct files 'config.h' and 'Makefile' out of templates, using a script to make the C compiler fit configure's expectations. The second command will compile and link gawk using the C compiler

directly; ignore any warnings from `make` about being unable to redefine `CC`. `configure` will take a very long time to execute, but at least it provides incremental feedback as it runs.

This has been tested with VAX/VMS V6.2, VMS POSIX V2.0, and DEC C V5.2.

Once built, `gawk` will work like any other shell utility. Unlike the normal VMS port of `gawk`, no special command line manipulation is needed in the VMS POSIX environment.

B.4 MS-DOS and OS/2 Installation and Compilation

If you have received a binary distribution prepared by the DOS maintainers, then `gawk` and the necessary support files will appear under the '`gnu`' directory, with executables in '`gnu/bin`', libraries in '`gnu/lib/awk`', and manual pages under '`gnu/man`'. This is designed for easy installation to a '`/gnu`' directory on your drive, but the files can be installed anywhere provided `AWKPATH` is set properly. Regardless of the installation directory, the first line of '`igawk.cmd`' and '`igawk.bat`' (in '`gnu/bin`') may need to be edited.

The binary distribution will contain a separate file describing the contents. In particular, it may include more than one version of the `gawk` executable. OS/2 binary distributions may have a different arrangement, but installation is similar.

The OS/2 and MS-DOS versions of `gawk` search for program files as described in Section 14.3 [The `AWKPATH` Environment Variable], page 164. However, semicolons (rather than colons) separate elements in the `AWKPATH` variable. If `AWKPATH` is not set or is empty, then the default search path is `".;c:/lib/awk;c:/gnu/lib/awk"`.

An `sh`-like shell (as opposed to `command.com` under MS-DOS or `cmd.exe` under OS/2) may be useful for `awk` programming. Ian Stewartson has written an excellent shell for MS-DOS and OS/2, and a `ksh` clone and GNU Bash are available for OS/2. The file '`README_d/README.pc`' in the `gawk` distribution contains information on these shells. Users of Stewartson's shell on DOS should examine its documentation on handling of command-lines. In particular, the setting for `gawk` in the shell configuration may need to be changed, and the `ignoretype` option may also be of interest.

`gawk` can be compiled for MS-DOS and OS/2 using the GNU development tools from DJ Delorie (DJGPP, MS-DOS-only) or Eberhard Mattes (EMX, MS-DOS and OS/2). Microsoft C can be used to build 16-bit versions for MS-DOS and OS/2. The file '`README_d/README.pc`' in the `gawk` distribution contains additional notes, and '`pc/Makefile`' contains important notes on compilation options.

To build gawk, copy the files in the 'pc' directory (*except* for 'ChangeLog') to the directory with the rest of the gawk sources. The 'Makefile' contains a configuration section with comments, and may need to be edited in order to work with your make utility.

The 'Makefile' contains a number of targets for building various MS-DOS and OS/2 versions. A list of targets will be printed if the make command is given without a target. As an example, to build gawk using the DJGPP tools, enter 'make djgpp'.

Using make to run the standard tests and to install gawk requires additional Unix-like tools, including sh, sed, and cp. In order to run the tests, the 'test/*.ok' files may need to be converted so that they have the usual DOS-style end-of-line markers. Most of the tests will work properly with Stewartson's shell along with the companion utilities or appropriate GNU utilities. However, some editing of 'test/Makefile' is required. It is recommended that the file 'pc/Makefile.tst' be copied to 'test/Makefile' as a replacement. Details can be found in 'README_d/README.pc'.

B.5 Installing gawk on the Atari ST

There are no substantial differences when installing gawk on various Atari models. Compiled gawk executables do not require a large amount of memory with most awk programs and should run on all Motorola processor based models (called further ST, even if that is not exactly right).

In order to use gawk, you need to have a shell, either text or graphics, that does not map all the characters of a command line to upper-case. Maintaining case distinction in option flags is very important (see Section 14.1 [Command Line Options], page 159). These days this is the default, and it may only be a problem for some very old machines. If your system does not preserve the case of option flags, you will need to upgrade your tools. Support for I/O redirection is necessary to make it easy to import awk programs from other environments. Pipes are nice to have, but not vital.

B.5.1 Compiling gawk on the Atari ST

A proper compilation of gawk sources when sizeof(int) differs from sizeof(void *) requires an ANSI C compiler. An initial port was done with gcc. You may actually prefer executables where ints are four bytes wide, but the other variant works as well.

You may need quite a bit of memory when trying to recompile the gawk sources, as some source files ('regex.c' in particular) are quite big. If you run out of memory compiling such a file, try reducing the optimization level for this particular file; this may help.

With a reasonable shell (Bash will do), and in particular if you run Linux, MiNT or a similar operating system, you have a pretty good chance that the configure utility will succeed. Otherwise sample versions of 'config.h'

and 'Makefile.st' are given in the 'atari' subdirectory and can be edited and copied to the corresponding files in the main source directory. Even if configure produced something, it might be advisable to compare its results with the sample versions and possibly make adjustments.

Some gawk source code fragments depend on a preprocessor define 'atarist'. This basically assumes the TOS environment with gcc. Modify these sections as appropriate if they are not right for your environment. Also see the remarks about AWKPATH and envsep in Section B.5.2 [Running gawk on the Atari ST], page 284.

As shipped, the sample 'config.h' claims that the system function is missing from the libraries, which is not true, and an alternative implementation of this function is provided in 'atari/system.c'. Depending upon your particular combination of shell and operating system, you may wish to change the file to indicate that system is available.

B.5.2 Running gawk on the Atari ST

An executable version of gawk should be placed, as usual, anywhere in your PATH where your shell can find it.

While executing, gawk creates a number of temporary files. When using gcc libraries for TOS, gawk looks for either of the environment variables TEMP or TMPDIR, in that order. If either one is found, its value is assumed to be a directory for temporary files. This directory must exist, and if you can spare the memory, it is a good idea to put it on a RAM drive. If neither TEMP nor TMPDIR are found, then gawk uses the current directory for its temporary files.

The ST version of gawk searches for its program files as described in Section 14.3 [The AWKPATH Environment Variable], page 164. The default value for the AWKPATH variable is taken from DEFPATH defined in 'Makefile'. The sample gcc/TOS 'Makefile' for the ST in the distribution sets DEFPATH to ".,c:\lib\awk,c:\gnu\lib\awk". The search path can be modified by explicitly setting AWKPATH to whatever you wish. Note that colons cannot be used on the ST to separate elements in the AWKPATH variable, since they have another, reserved, meaning. Instead, you must use a comma to separate elements in the path. When recompiling, the separating character can be modified by initializing the envsep variable in 'atari/gawkmisc.atr' to another value.

Although awk allows great flexibility in doing I/O redirections from within a program, this facility should be used with care on the ST running under TOS. In some circumstances the OS routines for file handle pool processing lose track of certain events, causing the computer to crash, and requiring a reboot. Often a warm reboot is sufficient. Fortunately, this happens infrequently, and in rather esoteric situations. In particular, avoid having one part of an awk program using print statements explicitly redirected

to "/dev/stdout", while other print statements use the default standard output, and a calling shell has redirected standard output to a file.

When gawk is compiled with the ST version of gcc and its usual libraries, it will accept both '/' and '\' as path separators. While this is convenient, it should be remembered that this removes one, technically valid, character ('/') from your file names, and that it may create problems for external programs, called via the system function, which may not support this convention. Whenever it is possible that a file created by gawk will be used by some other program, use only backslashes. Also remember that in awk, backslashes in strings have to be doubled in order to get literal backslashes (see Section 4.2 [Escape Sequences], page 24).

B.6 Installing gawk on an Amiga

You can install gawk on an Amiga system using a Unix emulation environment available via anonymous ftp from ftp.ninemoons.com in the directory 'pub/ade/current'. This includes a shell based on pdksh. The primary component of this environment is a Unix emulation library, 'ixemul.lib'.

A more complete distribution for the Amiga is available on the Geek Gadgets CD-ROM from:

CRONUS
1840 E. Warner Road #105-265
Tempe, AZ 85284 USA
US Toll Free: (800) 804-0833
Phone: +1-602-491-0442
FAX: +1-602-491-0048
Email: info@ninemoons.com
WWW: http://www.ninemoons.com
Anonymous ftp site: ftp.ninemoons.com

Once you have the distribution, you can configure gawk simply by running configure:

```
configure -v m68k-amigaos
```

Then run make, and you should be all set! (If these steps do not work, please send in a bug report; see Section B.7 [Reporting Problems and Bugs], page 285.)

B.7 Reporting Problems and Bugs

There is nothing more dangerous than a bored archeologist.
The Hitchhiker's Guide to the Galaxy

If you have problems with `gawk` or think that you have found a bug, please report it to the developers; we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you have actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible `awk` program and input data file that reproduces the problem. Then send us the program and data file, some idea of what kind of Unix system you're using, and the exact results `gawk` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you have a precise problem, send email to `bug-gawk@gnu.org`.

Please include the version number of `gawk` you are using. You can get this information with the command '`gawk --version`'. Using this address will automatically send a carbon copy of your mail to Arnold Robbins. If necessary, he can be reached directly at `arnold@gnu.org`.

Important! Do *not* try to report bugs in `gawk` by posting to the Usenet/Internet newsgroup `comp.lang.awk`. While the `gawk` developers do occasionally read this newsgroup, there is no guarantee that we will see your posting. The steps described above are the official, recognized ways for reporting bugs.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, ask Arnold Robbins; he will try to help you out, although he may not have the time to fix the problem. You can send him electronic mail at the Internet address above.

If you find bugs in one of the non-Unix ports of `gawk`, please send an electronic mail message to the person who maintains that port. They are listed below, and also in the '`README`' file in the `gawk` distribution. Information in the '`README`' file should be considered authoritative if it conflicts with this book.

The people maintaining the non-Unix ports of gawk are:

MS-DOS Scott Deifik, 'scottd@amgen.com', and Darrel Hankerson, 'hankedr@mail.auburn.edu'.

OS/2 Kai Uwe Rommel, 'rommel@ars.de'.

VMS Pat Rankin, 'rankin@eql.caltech.edu'.

Atari ST Michal Jaegermann, 'michal@gortel.phys.ualberta.ca'.

Amiga Fred Fish, 'fnf@ninemoons.com'.

If your bug is also reproducible under Unix, please send copies of your report to the general GNU bug list, as well as to Arnold Robbins, at the addresses listed above.

B.8 Other Freely Available awk Implementations

It's kind of fun to put comments like this in your awk code.

```
// Do C++ comments work? answer: yes! of course
```

Michael Brennan

There are two other freely available awk implementations. This section briefly describes where to get them.

Unix awk Brian Kernighan has been able to make his implementation of awk freely available. You can get it via anonymous ftp to the host netlib.bell-labs.com Change directory to '/netlib/research'. Use "binary" or "image" mode, and retrieve 'awk.bundle.gz'.

This is a shell archive that has been compressed with the GNU gzip utility. It can be uncompressed with the gunzip utility.

You can also retrieve this version via the World Wide Web from his home page (http://cm.bell-labs.com/who/bwk).

This version requires an ANSI C compiler; GCC (the GNU C compiler) works quite nicely.

mawk Michael Brennan has written an independent implementation of awk, called mawk. It is available under the GPL (see [GNU GENERAL PUBLIC LICENSE], page 303), just as gawk is.

You can get it via anonymous ftp to the host ftp.whidbey.net. Change directory to '/pub/brennan'. Use "binary" or "image" mode, and retrieve 'mawk1.3.3.tar.gz' (or the latest version that is there).

gunzip may be used to decompress this file. Installation is similar to gawk's (see Section B.2 [Compiling and Installing gawk on Unix], page 278).

Appendix C Implementation Notes

This appendix contains information mainly of interest to implementors and maintainers of `gawk`. Everything in it applies specifically to `gawk`, and not to other implementations.

C.1 Downward Compatibility and Debugging

See Section 17.5 [Extensions in `gawk` Not in POSIX `awk`], page 249, for a summary of the GNU extensions to the `awk` language and program. All of these features can be turned off by invoking `gawk` with the '`--traditional`' option, or with the '`--posix`' option.

If `gawk` is compiled for debugging with '`-DDEBUG`', then there is one more option available on the command line:

`-W parsedebug`
`--parsedebug`
: Print out the parse stack information as the program is being parsed.

This option is intended only for serious `gawk` developers, and not for the casual user. It probably has not even been compiled into your version of `gawk`, since it slows down execution.

C.2 Making Additions to `gawk`

If you should find that you wish to enhance `gawk` in a significant fashion, you are perfectly free to do so. That is the point of having free software; the source code is available, and you are free to change it as you wish (see [GNU GENERAL PUBLIC LICENSE], page 303).

This section discusses the ways you might wish to change `gawk`, and any considerations you should bear in mind.

C.2.1 Adding New Features

You are free to add any new features you like to `gawk`. However, if you want your changes to be incorporated into the `gawk` distribution, there are several steps that you need to take in order to make it possible for me to include your changes.

1. Get the latest version. It is much easier for me to integrate changes if they are relative to the most recent distributed version of `gawk`. If your version of `gawk` is very old, I may not be able to integrate them at all. See Section B.1.1 [Getting the `gawk` Distribution], page 273, for information on getting the latest version of `gawk`.

2. Follow the *GNU Coding Standards*. This document describes how GNU software should be written. If you haven't read it, please do so, preferably *before* starting to modify `gawk`. (The *GNU Coding Standards* are available as part of the Autoconf distribution, from the FSF.)
3. Use the `gawk` coding style. The C code for `gawk` follows the instructions in the *GNU Coding Standards*, with minor exceptions. The code is formatted using the traditional "K&R" style, particularly as regards the placement of braces and the use of tabs. In brief, the coding rules for `gawk` are:
   - Use old style (non-prototype) function headers when defining functions.
   - Put the name of the function at the beginning of its own line.
   - Put the return type of the function, even if it is `int`, on the line above the line with the name and arguments of the function.
   - The declarations for the function arguments should not be indented.
   - Put spaces around parentheses used in control structures (`if`, `while`, `for`, `do`, `switch` and `return`).
   - Do not put spaces in front of parentheses used in function calls.
   - Put spaces around all C operators, and after commas in function calls.
   - Do not use the comma operator to produce multiple side-effects, except in `for` loop initialization and increment parts, and in macro bodies.
   - Use real tabs for indenting, not spaces.
   - Use the "K&R" brace layout style.
   - Use comparisons against `NULL` and `'\0'` in the conditions of `if`, `while` and `for` statements, and in the `case`s of `switch` statements, instead of just the plain pointer or character value.
   - Use the `TRUE`, `FALSE`, and `NULL` symbolic constants, and the character constant `'\0'` where appropriate, instead of `1` and `0`.
   - Provide one-line descriptive comments for each function.
   - Do not use '`#elif`'. Many older Unix C compilers cannot handle it.
   - Do not use the `alloca` function for allocating memory off the stack. Its use causes more portability trouble than the minor benefit of not having to free the storage. Instead, use `malloc` and `free`.

   If I have to reformat your code to follow the coding style used in `gawk`, I may not bother.
4. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your changes, you must either place those changes in the public domain, and submit a signed statement to that effect, or assign the

copyright in your changes to the FSF. Both of these actions are easy to do, and *many* people have done so already. If you have questions, please contact me (see Section B.7 [Reporting Problems and Bugs], page 285), or `gnu@gnu.org`.

5. Update the documentation. Along with your new code, please supply new sections and or chapters for this book. If at all possible, please use real Texinfo, instead of just supplying unformatted ASCII text (although even that is better than no documentation at all). Conventions to be followed in *Effective AWK Programming* are provided after the '`@bye`' at the end of the Texinfo source file. If possible, please update the man page as well.

   You will also have to sign paperwork for your documentation changes.

6. Submit changes as context diffs or unified diffs. Use '`diff -c -r -N`' or '`diff -u -r -N`' to compare the original `gawk` source tree with your version. (I find context diffs to be more readable, but unified diffs are more compact.) I recommend using the GNU version of `diff`. Send the output produced by either run of `diff` to me when you submit your changes. See Section B.7 [Reporting Problems and Bugs], page 285, for the electronic mail information.

   Using this format makes it easy for me to apply your changes to the master version of the `gawk` source code (using `patch`). If I have to apply the changes manually, using a text editor, I may not do so, particularly if there are lots of changes.

7. Include an entry for the '`ChangeLog`' file with your submission. This further helps minimize the amount of work I have to do, making it easier for me to accept patches.

Although this sounds like a lot of work, please remember that while you may write the new code, I have to maintain it and support it, and if it isn't possible for me to do that with a minimum of extra work, then I probably will not.

C.2.2 Porting `gawk` to a New Operating System

If you wish to port `gawk` to a new operating system, there are several steps to follow.

1. Follow the guidelines in Section C.2.1 [Adding New Features], page 289, concerning coding style, submission of diffs, and so on.
2. When doing a port, bear in mind that your code must co-exist peacefully with the rest of `gawk`, and the other ports. Avoid gratuitous changes to the system-independent parts of the code. If at all possible, avoid sprinkling '`#ifdef`'s just for your port throughout the code.

   If the changes needed for a particular system affect too much of the code, I probably will not accept them. In such a case, you will, of course, be

able to distribute your changes on your own, as long as you comply with the GPL (see [GNU GENERAL PUBLIC LICENSE], page 303).

3. A number of the files that come with `gawk` are maintained by other people at the Free Software Foundation. Thus, you should not change them unless it is for a very good reason. I.e. changes are not out of the question, but changes to these files will be scrutinized extra carefully. The files are '`alloca.c`', '`getopt.h`', '`getopt.c`', '`getopt1.c`', '`regex.h`', '`regex.c`', '`dfa.h`', '`dfa.c`', '`install-sh`', and '`mkinstalldirs`'.
4. Be willing to continue to maintain the port. Non-Unix operating systems are supported by volunteers who maintain the code needed to compile and run `gawk` on their systems. If no-one volunteers to maintain a port, that port becomes unsupported, and it may be necessary to remove it from the distribution.
5. Supply an appropriate '`gawkmisc.???`' file. Each port has its own '`gawkmisc.???`' that implements certain operating system specific functions. This is cleaner than a plethora of '`#ifdef`'s scattered throughout the code. The '`gawkmisc.c`' in the main source directory includes the appropriate '`gawkmisc.???`' file from each subdirectory. Be sure to update it as well.

   Each port's '`gawkmisc.???`' file has a suffix reminiscent of the machine or operating system for the port. For example, '`pc/gawkmisc.pc`' and '`vms/gawkmisc.vms`'. The use of separate suffixes, instead of plain '`gawkmisc.c`', makes it possible to move files from a port's subdirectory into the main subdirectory, without accidentally destroying the real '`gawkmisc.c`' file. (Currently, this is only an issue for the MS-DOS and OS/2 ports.)
6. Supply a '`Makefile`' and any other C source and header files that are necessary for your operating system. All your code should be in a separate subdirectory, with a name that is the same as, or reminiscent of, either your operating system or the computer system. If possible, try to structure things so that it is not necessary to move files out of the subdirectory into the main source directory. If that is not possible, then be sure to avoid using names for your files that duplicate the names of files in the main source directory.
7. Update the documentation. Please write a section (or sections) for this book describing the installation and compilation steps needed to install and/or compile `gawk` for your system.
8. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your code, you must either place your code in the public domain, and submit a signed statement to that effect, or assign the copyright in your code to the FSF.

Following these steps will make it much easier to integrate your changes into `gawk`, and have them co-exist happily with the code for other operating systems that is already there.

In the code that you supply, and that you maintain, feel free to use a coding style and brace layout that suits your taste.

C.3 Probable Future Extensions

> *AWK is a language similar to PERL, only considerably more elegant.*
> Arnold Robbins
>
> *Hey!*
> Larry Wall

This section briefly lists extensions and possible improvements that indicate the directions we are currently considering for `gawk`. The file '`FUTURES`' in the `gawk` distributions lists these extensions as well.

This is a list of probable future changes that will be usable by the `awk` language programmer.

Localization
: The GNU project is starting to support multiple languages. It will at least be possible to make `gawk` print its warnings and error messages in languages other than English. It may be possible for `awk` programs to also use the multiple language facilities, separate from `gawk` itself.

Databases
: It may be possible to map a GDBM/NDBM/SDBM file into an `awk` array.

A `PROCINFO` Array
: The special files that provide process-related information (see Section 6.7 [Special File Names in `gawk`], page 69) will be superseded by a `PROCINFO` array that would provide the same information, in an easier to access fashion.

More `lint` warnings
: There are more things that could be checked for portability.

Control of subprocess environment
: Changes made in `gawk` to the array `ENVIRON` may be propagated to subprocesses run by `gawk`.

This is a list of probable improvements that will make `gawk` perform better.

An Improved Version of `dfa`
: The `dfa` pattern matcher from GNU `grep` has some problems. Either a new version or a fixed one will deal with some important regexp matching issues.

Use of GNU `malloc`
: The GNU version of `malloc` could potentially speed up `gawk`, since it relies heavily on the use of dynamic memory allocation.

C.4 Suggestions for Improvements

Here are some projects that would-be `gawk` hackers might like to take on. They vary in size from a few days to a few weeks of programming, depending on which one you choose and how fast a programmer you are. Please send any improvements you write to the maintainers at the GNU project. See Section C.2.1 [Adding New Features], page 289, for guidelines to follow when adding new features to `gawk`. See Section B.7 [Reporting Problems and Bugs], page 285, for information on contacting the maintainers.

1. Compilation of `awk` programs: `gawk` uses a Bison (YACC-like) parser to convert the script given it into a syntax tree; the syntax tree is then executed by a simple recursive evaluator. This method incurs a lot of overhead, since the recursive evaluator performs many procedure calls to do even the simplest things.

   It should be possible for `gawk` to convert the script's parse tree into a C program which the user would then compile, using the normal C compiler and a special `gawk` library to provide all the needed functions (regexps, fields, associative arrays, type coercion, and so on).

   An easier possibility might be for an intermediate phase of `awk` to convert the parse tree into a linear byte code form like the one used in GNU Emacs Lisp. The recursive evaluator would then be replaced by a straight line byte code interpreter that would be intermediate in speed between running a compiled program and doing what `gawk` does now.
2. The programs in the test suite could use documenting in this book.
3. See the '`FUTURES`' file for more ideas. Contact us if you would seriously like to tackle any of the items listed there.

Appendix D Glossary

Action
: A series of `awk` statements attached to a rule. If the rule's pattern matches an input record, `awk` executes the rule's action. Actions are always enclosed in curly braces. See Section 8.2 [Overview of Actions], page 100.

Amazing `awk` Assembler
: Henry Spencer at the University of Toronto wrote a retargetable assembler completely as `awk` scripts. It is thousands of lines long, including machine descriptions for several eight-bit microcomputers. It is a good example of a program that would have been better written in another language.

Amazingly Workable Formatter (`awf`)
: Henry Spencer at the University of Toronto wrote a formatter that accepts a large subset of the '`nroff -ms`' and '`nroff -man`' formatting commands, using `awk` and `sh`.

ANSI
: The American National Standards Institute. This organization produces many standards, among them the standards for the C and C++ programming languages.

Assignment
: An `awk` expression that changes the value of some `awk` variable or data object. An object that you can assign to is called an *lvalue*. The assigned values are called *rvalues*. See Section 7.7 [Assignment Expressions], page 82.

`awk` Language
: The language in which `awk` programs are written.

`awk` Program
: An `awk` program consists of a series of *patterns* and *actions*, collectively known as *rules*. For each input record given to the program, the program's rules are all processed in turn. `awk` programs may also contain function definitions.

`awk` Script
: Another name for an `awk` program.

Bash
: The GNU version of the standard shell (the Bourne-Again shell). See "Bourne Shell."

BBS
: See "Bulletin Board System."

Boolean Expression
: Named after the English mathematician Boole. See "Logical Expression."

Bourne Shell
: The standard shell ('`/bin/sh`') on Unix and Unix-like systems, originally written by Steven R. Bourne. Many shells (Bash, `ksh`,

`pdksh`, `zsh`) are generally upwardly compatible with the Bourne shell.

Built-in Function
: The `awk` language provides built-in functions that perform various numerical, time stamp related, and string computations. Examples are `sqrt` (for the square root of a number) and `substr` (for a substring of a string). See Chapter 12 [Built-in Functions], page 133.

Built-in Variable
: `ARGC`, `ARGIND`, `ARGV`, `CONVFMT`, `ENVIRON`, `ERRNO`, `FIELDWIDTHS`, `FILENAME`, `FNR`, `FS`, `IGNORECASE`, `NF`, `NR`, `OFMT`, `OFS`, `ORS`, `RLENGTH`, `RSTART`, `RS`, `RT`, and `SUBSEP`, are the variables that have special meaning to `awk`. Changing some of them affects `awk`'s running environment. Several of these variables are specific to `gawk`. See Chapter 10 [Built-in Variables], page 113.

Braces
: See "Curly Braces."

Bulletin Board System
: A computer system allowing users to log in and read and/or leave messages for other users of the system, much like leaving paper notes on a bulletin board.

C
: The system programming language that most GNU software is written in. The `awk` programming language has C-like syntax, and this book points out similarities between `awk` and C when appropriate.

Character Set
: The set of numeric codes used by a computer system to represent the characters (letters, numbers, punctuation, etc.) of a particular country or place. The most common character set in use today is ASCII (American Standard Code for Information Interchange). Many European countries use an extension of ASCII known as ISO-8859-1 (ISO Latin-1).

CHEM
: A preprocessor for `pic` that reads descriptions of molecules and produces `pic` input for drawing them. It was written in `awk` by Brian Kernighan and Jon Bentley, and is available from `netlib@research.bell-labs.com`

Compound Statement
: A series of `awk` statements, enclosed in curly braces. Compound statements may be nested. See Chapter 9 [Control Statements in Actions], page 103.

Concatenation
: Concatenating two strings means sticking them together, one after another, giving a new string. For example, the string '`foo`'

concatenated with the string '`bar`' gives the string '`foobar`'. See Section 7.6 [String Concatenation], page 81.

Conditional Expression
: An expression using the '`?:`' ternary operator, such as '*expr1* `?` *expr2* `:` *expr3*'. The expression *expr1* is evaluated; if the result is true, the value of the whole expression is the value of *expr2*, otherwise the value is *expr3*. In either case, only one of *expr2* and *expr3* is evaluated. See Section 7.12 [Conditional Expressions], page 90.

Comparison Expression
: A relation that is either true or false, such as '`(a < b)`'. Comparison expressions are used in `if`, `while`, `do`, and `for` statements, and in patterns to select which input records to process. See Section 7.10 [Variable Typing and Comparison Expressions], page 86.

Curly Braces
: The characters '`{`' and '`}`'. Curly braces are used in `awk` for delimiting actions, compound statements, and function bodies.

Dark Corner
: An area in the language where specifications often were (or still are) not clear, leading to unexpected or undesirable behavior. Such areas are marked in this book with "(d.c.)" in the text, and are indexed under the heading "dark corner."

Data Objects
: These are numbers and strings of characters. Numbers are converted into strings and vice versa, as needed. See Section 7.4 [Conversion of Strings and Numbers], page 79.

Double Precision
: An internal representation of numbers that can have fractional parts. Double precision numbers keep track of more digits than do single precision numbers, but operations on them are more expensive. This is the way `awk` stores numeric values. It is the C type `double`.

Dynamic Regular Expression
: A dynamic regular expression is a regular expression written as an ordinary expression. It could be a string constant, such as `"foo"`, but it may also be an expression whose value can vary. See Section 4.7 [Using Dynamic Regexps], page 35.

Environment
: A collection of strings, of the form *name*=*val*, that each program has available to it. Users generally place values into the environment in order to provide information to various programs. Typical examples are the environment variables `HOME` and `PATH`.

Empty String
: See "Null String."

Escape Sequences
: A special sequence of characters used for describing non-printing characters, such as '`\n`' for newline, or '`\033`' for the ASCII ESC (escape) character. See Section 4.2 [Escape Sequences], page 24.

Field
: When `awk` reads an input record, it splits the record into pieces separated by whitespace (or by a separator regexp which you can change by setting the built-in variable `FS`). Such pieces are called fields. If the pieces are of fixed length, you can use the built-in variable `FIELDWIDTHS` to describe their lengths. See Section 5.5 [Specifying How Fields are Separated], page 44, and also see See Section 5.6 [Reading Fixed-width Data], page 49.

Floating Point Number
: Often referred to in mathematical terms as a "rational" number, this is just a number that can have a fractional part. See "Double Precision" and "Single Precision."

Format
: Format strings are used to control the appearance of output in the `printf` statement. Also, data conversions from numbers to strings are controlled by the format string contained in the built-in variable `CONVFMT`. See Section 6.5.2 [Format-Control Letters], page 63.

Function
: A specialized group of statements used to encapsulate general or program-specific tasks. `awk` has a number of built-in functions, and also allows you to define your own. See Chapter 12 [Built-in Functions], page 133, and Chapter 13 [User-defined Functions], page 151.

FSF
: See "Free Software Foundation."

Free Software Foundation
: A non-profit organization dedicated to the production and distribution of freely distributable software. It was founded by Richard M. Stallman, the author of the original Emacs editor. GNU Emacs is the most widely used version of Emacs today.

`gawk`
: The GNU implementation of `awk`.

General Public License
: This document describes the terms under which `gawk` and its source code may be distributed. (see [GNU GENERAL PUBLIC LICENSE], page 303)

GNU
: "GNU's not Unix". An on-going project of the Free Software Foundation to create a complete, freely distributable, POSIX-compliant computing environment.

GPL — See "General Public License."

Hexadecimal
: Base 16 notation, where the digits are `0-9` and `A-F`, with '`A`' representing 10, '`B`' representing 11, and so on up to '`F`' for 15. Hexadecimal numbers are written in C using a leading '`0x`', to indicate their base. Thus, `0x12` is 18 (one times 16 plus 2).

I/O — Abbreviation for "Input/Output," the act of moving data into and/or out of a running program.

Input Record
: A single chunk of data read in by `awk`. Usually, an `awk` input record consists of one line of text. See Section 5.1 [How Input is Split into Records], page 37.

Integer — A whole number, i.e. a number that does not have a fractional part.

Keyword — In the `awk` language, a keyword is a word that has special meaning. Keywords are reserved and may not be used as variable names.

`gawk`'s keywords are: `BEGIN`, `END`, `if`, `else`, `while`, `do...while`, `for`, `for...in`, `break`, `continue`, `delete`, `next`, `nextfile`, `function`, `func`, and `exit`.

Logical Expression
: An expression using the operators for logic, AND, OR, and NOT, written '`&&`', '`||`', and '`!`' in `awk`. Often called Boolean expressions, after the mathematician who pioneered this kind of mathematical logic.

Lvalue — An expression that can appear on the left side of an assignment operator. In most languages, lvalues can be variables or array elements. In `awk`, a field designator can also be used as an lvalue.

Null String
: A string with no characters in it. It is represented explicitly in `awk` programs by placing two double-quote characters next to each other (`""`). It can appear in input data by having two successive occurrences of the field separator appear next to each other.

Number — A numeric valued data object. The `gawk` implementation uses double precision floating point to represent numbers. Very old `awk` implementations use single precision floating point.

Octal — Base-eight notation, where the digits are `0-7`. Octal numbers are written in C using a leading '`0`', to indicate their base. Thus, `013` is 11 (one times 8 plus 3).

Pattern
: Patterns tell `awk` which input records are interesting to which rules.

  A pattern is an arbitrary conditional expression against which input is tested. If the condition is satisfied, the pattern is said to *match* the input record. A typical pattern might compare the input record against a regular expression. See Section 8.1 [Pattern Elements], page 95.

POSIX
: The name for a series of standards being developed by the IEEE that specify a Portable Operating System interface. The "IX" denotes the Unix heritage of these standards. The main standard of interest for `awk` users is *IEEE Standard for Information Technology, Standard 1003.2-1992, Portable Operating System Interface (POSIX) Part 2: Shell and Utilities*. Informally, this standard is often referred to as simply "P1003.2."

Private
: Variables and/or functions that are meant for use exclusively by library functions, and not for the main `awk` program. Special care must be taken when naming such variables and functions. See Section 15.13 [Naming Library Function Global Variables], page 199.

Range (of input lines)
: A sequence of consecutive lines from the input file. A pattern can specify ranges of input lines for `awk` to process, or it can specify single lines. See Section 8.1 [Pattern Elements], page 95.

Recursion
: When a function calls itself, either directly or indirectly. If this isn't clear, refer to the entry for "recursion."

Redirection
: Redirection means performing input from other than the standard input stream, or output to other than the standard output stream.

  You can redirect the output of the `print` and `printf` statements to a file or a system command, using the '`>`', '`>>`', and '`|`' operators. You can redirect input to the `getline` statement using the '`<`' and '`|`' operators. See Section 6.6 [Redirecting Output of `print` and `printf`], page 67, and Section 5.8 [Explicit Input with `getline`], page 53.

Regexp
: Short for *regular expression*. A regexp is a pattern that denotes a set of strings, possibly an infinite set. For example, the regexp '`R.*xp`' matches any string starting with the letter '`R`' and ending with the letters '`xp`'. In `awk`, regexps are used in patterns and in conditional expressions. Regexps may contain escape sequences. See Chapter 4 [Regular Expressions], page 23.

Regular Expression
: See "regexp."

Regular Expression Constant
: A regular expression constant is a regular expression written within slashes, such as `/foo/`. This regular expression is chosen when you write the `awk` program, and cannot be changed doing its execution. See Section 4.1 [How to Use Regular Expressions], page 23.

Rule
: A segment of an `awk` program that specifies how to process single input records. A rule consists of a *pattern* and an *action*. `awk` reads an input record; then, for each rule, if the input record satisfies the rule's pattern, `awk` executes the rule's action. Otherwise, the rule does nothing for that input record.

Rvalue
: A value that can appear on the right side of an assignment operator. In `awk`, essentially every expression has a value. These values are rvalues.

`sed`
: See "Stream Editor."

Short-Circuit
: The nature of the `awk` logical operators '`&&`' and '`||`'. If the value of the entire expression can be deduced from evaluating just the left-hand side of these operators, the right-hand side will not be evaluated (see Section 7.11 [Boolean Expressions], page 89).

Side Effect
: A side effect occurs when an expression has an effect aside from merely producing a value. Assignment expressions, increment and decrement expressions and function calls have side effects. See Section 7.7 [Assignment Expressions], page 82.

Single Precision
: An internal representation of numbers that can have fractional parts. Single precision numbers keep track of fewer digits than do double precision numbers, but operations on them are less expensive in terms of CPU time. This is the type used by some very old versions of `awk` to store numeric values. It is the C type `float`.

Space
: The character generated by hitting the space bar on the keyboard.

Special File
: A file name interpreted internally by `gawk`, instead of being handed directly to the underlying operating system. For example, '`/dev/stderr`'. See Section 6.7 [Special File Names in `gawk`], page 69.

Stream Editor
: A program that reads records from an input stream and processes them one or more at a time. This is in contrast with batch

programs, which may expect to read their input files in entirety before starting to do anything, and with interactive programs, which require input from the user.

String — A datum consisting of a sequence of characters, such as '`I am a string`'. Constant strings are written with double-quotes in the `awk` language, and may contain escape sequences. See Section 4.2 [Escape Sequences], page 24.

Tab — The character generated by hitting the *TAB* key on the keyboard. It usually expands to up to eight spaces upon output.

Unix — A computer operating system originally developed in the early 1970's at AT&T Bell Laboratories. It initially became popular in universities around the world, and later moved into commercial evnironments as a software development system and network server system. There are many commercial versions of Unix, as well as several work-alike systems whose source code is freely available (such as Linux, NetBSD, and FreeBSD).

Whitespace — A sequence of space, tab, or newline characters occurring inside an input record or a string.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place — Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary.

To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an an-

nouncement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for non-commercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source

code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN

OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and an idea of what it does.*
> Copyright (C) *year name of author*
>
> This program is free software; you can redistribute it and/or
> modify it under the terms of the GNU General Public License
> as published by the Free Software Foundation; either version 2
> of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful,
> but WITHOUT ANY WARRANTY; without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
> GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License
> along with this program; if not, write to the Free Software
> Foundation, Inc., 59 Temple Place --- Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) *year name of author*
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
> type 'show w'. This is free software, and you are welcome
> to redistribute it under certain conditions; type 'show c'
> for details.

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

!

! operator 89
!= operator 87
!~ operator 23, 33, 35, 75, 87

#

(comment) 13
#! (executable scripts) 12

$

$ (field operator) 40

&

&& operator 89

-

--assign option 159
--compat option 160
--copyleft option 160
--copyright option 160
--field-separator option 159
--file option 159
--help option 161
--lint option 161
--lint-old option 161
--posix option 161
--source option 162
--traditional option 160
--usage option 161
--version option 162
-f option 11, 159
-F option 46, 159
-v option 159
-W option 160

/

'/dev/fd' 70
'/dev/pgrpid' 70
'/dev/pid' 70
'/dev/ppid' 70
'/dev/stderr' 70
'/dev/stdin' 70
'/dev/stdout' 70
'/dev/user' 70, 189

=

== operator 87

_

_gr_init 195
_pw_init 191
_tm_addup 177
_tm_isleap 176

|

|| operator 89

~

~ operator 23, 33, 35, 75, 87

>

> operator 87
>= operator 87

\

\' regexp operator 32
\` regexp operator 32
\> regexp operator 32
\< regexp operator 32
\B regexp operator 32
\w regexp operator 31
\W regexp operator 31
\y regexp operator 32

<

< operator 87
<= operator 87

A

accessing fields 40
account information 189, 194
acronym 1
action, curly braces 100
action, default 14
action, definition of 100
action, empty 14
action, separating statements 100
adding new features 289
addition 80
Aho, Alfred 1
AI programming, using `gawk` 276
`alarm.awk` 224
amiga 285
anchors in regexps 27
and operator 89
anonymous `ftp` 273, 287
applications of `awk` 18
`ARGC` 115
`ARGIND` 115, 163
argument processing 184
arguments in function call 91
arguments, command line 159
`ARGV` 115, 163
arithmetic operators 80
array assignment 123
array reference 122
Array subscripts and `IGNORECASE` 122
array subscripts, uninitialized variables 127
arrays 121
arrays, associative 121
arrays, definition of 121
arrays, deleting an element 125
arrays, deleting entire contents 126
arrays, multi-dimensional subscripts 128
arrays, presence of elements 123
arrays, sparse 122
arrays, special `for` statement 124
arrays, the `in` operator 123
artificial intelligence, using `gawk` 276
ASCII 173
`assert` 170
`assert`, C version 169
assertions 169
assignment operators 82
assignment to fields 42
associative arrays 121
`atan2` 134
atari 283
automatic initialization 16
`awk` language, POSIX version 25, 27, 28, 30, 48, 62, 66, 79, 80, 84, 93, 107, 108, 109, 113, 136, 142, 152
`awk` language, V.4 version 24, 25, 248
`AWKPATH` environment variable 164
`awksed` 237

B

backslash continuation 16, 210
backslash continuation and comments 17
backslash continuation in `csh` 15, 17
basic function of `awk` 9
'`BBS-list`' file 7
`BEGIN` special pattern 98
`beginfile` 183
body of a loop 104
book, using this 5
boolean expressions 89
boolean operators 89
`break` statement 106
`break`, outside of loops 107
Brennan, Michael 126, 237, 287
buffer matching operators 32
buffering output 144, 145
buffering, interactive vs. non-interactive 145
buffering, non-interactive vs. interactive 145
buffers, flushing 144, 145
bugs, known in `gawk` 165
built-in functions 133
built-in variables 113
built-in variables, convey information 115
built-in variables, user modifiable 113

C

call by reference 154, 155
call by value 154
calling a function 91, 154
case conversion 141
case sensitivity 33
changing contents of a field 42
changing the record separator 37
character classes 28
character encodings 173
character list 27
character list, complemented 29
character sets 173
`chr` 172
`close` 71, 143
closing input files and pipes 71
closing output files and pipes 71
coding style used in `gawk` 290
collating elements 29
collating symbols 29
command line 159
command line formats 10
command line, setting `FS` on 46
comments 13
comments and backslash continuation .. 17
common mistakes 35, 44, 60, 87
`comp.lang.awk` 286
comparison expressions 86
comparisons, string vs. regexp 88
compatibility mode 160, 249
complemented character list 29
compound statement 103
computed regular expressions 35
concatenation 81
conditional expression 90
configuring `gawk` 279
constants, types of 75
continuation of lines 16
`continue` statement 107
`continue`, outside of loops 108
control statement 103
conversion of case 141
conversion of strings and numbers 79
conversions, during subscripting 127
converting dates to timestamps 175
`CONVFMT` 79, 113, 127
`cos` 134
`csh`, backslash continuation 15, 17
curly braces 100
`custom.h` configuration file 279
`cut` utility 201
`cut.awk` 202

D

d.c., see "dark corner" 6
dark corner .. 6, 26, 39, 46, 48, 51, 54, 58, 62, 64, 65, 76, 78, 79, 86, 99, 107, 108, 110, 116, 117, 128, 137, 159, 163
data-driven languages 9
dates, converting to timestamps 175
decrement operators 85
default action 14
default pattern 14
defining functions 151
Deifik, Scott 3, 287
`delete` statement 125
deleting elements of arrays 125
deleting entire arrays 126
deprecated features 165
deprecated options 165
differences between `gawk` and `awk` .. 33, 39, 46, 53, 58, 69, 74, 75, 76, 81, 91, 100, 109, 119, 126, 133, 137, 140, 164
directory search 164
division 80
documenting `awk` programs 13, 199
`dupword.awk` 223
dynamic regular expressions 35

E

EBCDIC 173
`egrep` 10, 28
`egrep` utility 206
`egrep.awk` 207
element assignment 123
element of array 122
empty action 14
empty pattern 100
empty program 159
empty string 39, 46, 79, 86

`END` special pattern 98
`endfile` 183
`endgrent` 198
`endpwent` 193
`ENVIRON` 116
environment variable, `AWKPATH` 164
environment variable, `POSIXLY_CORRECT` 162
equivalence classes 29
`ERRNO` 53, 74, 116
errors, common 35, 44, 60, 87
escape processing, `sub` et. al. 141
escape sequence notation 24
evaluation, order of 133
examining fields 40
executable scripts 12
`exit` statement 110
`exp` 134
explicit input 53
exponentiation 80
expression 75
expression, assignment 82
expression, boolean 89
expression, comparison 86
expression, conditional 90
expression, matching 86
`extract.awk` 234

F

features, adding 289
`fflush` 144
field operator `$` 40
field separator, choice of 44
field separator, `FS` 44
field separator, on command line 46
field, changing contents of 42
fields 40
fields, separating 44
`FIELDWIDTHS` 113
file descriptors 69
file, `awk` program 11
`FILENAME` 37, 58, 116
`FILENAME`, being set by `getline` 58
Fish, Fred 287
flushing buffers 144, 145
`FNR` 40, 116
`for (x in ...)` 124
`for` statement 105
format specifier 63
format string 62
format, numeric output 62
formatted output 62
formatted timestamps 180
Free Software Foundation 1, 273
FreeBSD 2
Friedl, Jeffrey 4
`FS` 44, 113
`ftp`, anonymous 273, 287
function call 91, 154
function definition 151
function, recursive 152
functions, undefined 155
functions, user-defined 151

G

`gawk` coding style 290
`gensub` 139
`getgrent` 198
`getgrent`, C version 194
`getgrgid` 197
`getgrnam` 197
`getgruser` 198
`getline` 53
`getline`, return values 53
`getline`, setting `FILENAME` 58
`getopt` 186
`getopt`, C version 184
`getpwent` 193
`getpwent`, C version 189
`getpwnam` 192
`getpwuid` 193
`gettimeofday` 180
getting `gawk` 273
GNU Project 1
`grcat` program 194
`grcat.c` 194
group file 194
group information 194
`gsub` 139
`gsub`, third argument of 139

H

Hankerson, Darrel 3, 287
historical features .. 47, 107, 108, 136, 271
history of `awk` 1
`histsort.awk` 233
how `awk` works 14
Hughes, Phil 4

I

I/O from `BEGIN` and `END` 99
`id` utility 210
`id.awk` 211
`if-else` statement 103
`igawk.sh` 241
`IGNORECASE` 33, 114, 122
`IGNORECASE` and array subscripts 122
ignoring case 33
implementation limits 58, 69
`in` operator 87
increment operators 85
`index` 135
initialization, automatic 16
input 37
input file, sample 7
input files, skipping 167
input pipeline 56
input redirection 55
input, explicit 53
input, `getline` command 53
input, multiple line records 50
input, standard 11
installation, amiga 285
installation, atari 283
installation, MS-DOS and OS/2 282
installation, unix 278
installation, vms 279
`int` 134
interaction, `awk` and other programs .. 144
interactive buffering vs. non-interactive 145
interval expressions 30
'`inventory-shipped`' file 7
invocation of `gawk` 159
ISO 8601 149
ISO 8859-1 ..... 34, 296
ISO Latin-1 ..... 34, 296

J

Jaegermann, Michal 3, 287
`join` 174

K

Kernighan, Brian 1, 4, 81, 249, 287
known bugs 165

L

`labels.awk` 229
language, `awk` 5
language, data-driven 9
language, procedural 9
leftmost longest match 34, 51
`length` 136
limitations 58, 69
line break 16
line continuation 16, 61, 90, 91
Linux 2, 283
locale, definition of 148
`log` 134
logical false 86
logical operations 89
logical true 86
login information 189
long options 159
loop 104
loops, exiting 106
lvalue 82

M

mark parity 173
`match` 136
matching ranges of lines 97
matching, leftmost longest 34, 51
`mawk` 287
merging strings 174
metacharacters 26
mistakes, common 35, 44, 60, 87
`mktime` 178

modifiers (in format specifiers) 64
multi-dimensional subscripts 128
multiple line records 50
multiple passes over data 164
multiple statements on one line 18
multiplication 80

N

names, use of 151
namespace issues in `awk` 199
namespaces 151
NetBSD 2
new `awk` 1
new `awk` vs. old `awk` 9
newline 16
`next file` statement 110
`next` statement 108
`next`, inside a user-defined function .. 109
`nextfile` function 168
`nextfile` statement 109
`NF` 40, 116
non-interactive buffering vs. interactive 145
not operator 89
`NR` 40, 116
null string 46, 79, 86
null string, as array subscript 128
number of fields, `NF` 40
number of records, `NR`, `FNR` 40
numbers, used as subscripts 127
numeric character values 172
numeric constant 75
numeric output format 62
numeric string 86
numeric value 75

O

obsolete features 165
obsolete options 165
`OFMT` 62, 79, 114
`OFS` 61, 114
old `awk` 1
old `awk` vs. new `awk` 9
one-liners 21
operations, logical 89
operator precedence 92
operators, arithmetic 80
operators, assignment 82
operators, boolean 89
operators, decrement 85
operators, increment 85
operators, regexp matching 23
operators, relational 86, 87
operators, short-circuit 89
operators, string 81
operators, string-matching 23
options, command line 159
options, long 159
or operator 89
`ord` 172
order of evaluation 133
`ORS` 61, 114
output 59
output field separator, `OFS` 61
output format specifier, `OFMT` 62
output record separator, `ORS` 61
output redirection 67
output, buffering 144, 145
output, formatted 62
output, piping 68

P

passes, multiple 164
password file 189
path, search 164
pattern, `BEGIN` 98
pattern, default 14
pattern, definition of 95
pattern, empty 100
pattern, `END` 98
pattern, range 97
pattern, regular expressions 23
patterns, types of 95
per file initialization and clean-up 182
PERL 293
pipeline, input 56
pipes for output 68
portability issues 17, 25, 74, 126, 136, 144, 152, 167

porting gawk 291
POSIX awk 25, 27, 28, 30, 48, 62, 66, 79, 80, 84, 93, 107, 108, 109, 113, 136, 142, 152
POSIX mode 161
POSIXLY_CORRECT environment variable 162
precedence 92
precedence, regexp operators 31
print statement 59
printf statement, syntax of 62
printf, format-control characters 63
printf, modifiers 64
printing 59
procedural languages 9
process information 70
processing arguments 184
program file 11
program, awk 5
program, definition of 9
program, self contained 12
programs, documenting 13, 199
pwcat program 189
pwcat.c 190

Q

quotient 80
quoting, shell 11, 12

R

Rakitzis, Byron 233
rand 134
random numbers, seed of 135
range pattern 97
Rankin, Pat 3, 83, 287
reading files 37
reading files, getline command 53
reading files, multiple line records 50
record separator, RS 37
record terminator, RT 39
record, definition of 37
records, multiple line 50
recursive function 152
redirection of input 55
redirection of output 67
reference to array 122
regexp 23
regexp as expression 89
regexp comparison vs. string comparison 88
regexp constant 24
regexp constants, difference between slashes and quotes 35
regexp match/non-match operators ... 23, 86
regexp matching operators 23
regexp operators 26
regexp operators, GNU specific 31
regexp operators, precedence of 31
regexp, anchors 27
regexp, dynamic 35
regexp, effect of command line options 32
regular expression 23
regular expression metacharacters 26
regular expressions as field separators .. 44
regular expressions as patterns 23
regular expressions as record separators 39
regular expressions, computed 35
relational operators 86, 87
remainder 80
removing elements of arrays 125
return statement 156
RFC-1036 149
RFC-822 149
RLENGTH 116, 136
Robbins, Miriam 4
Rommel, Kai Uwe 3, 287
round 171
rounding 171
RS 37, 114
RSTART 117, 136
RT 39, 53, 117
rule, definition of 9
running awk programs 10
running long programs 11
rvalue 82

S

sample input file 7
scanning an array 124
script, definition of 9
scripts, executable 12
scripts, shell 12
search path 164
search path, for source files 164
`sed` utility 48, 237, 241
seed for random numbers 135
self contained programs 12
shell quoting 11, 12
shell scripts 12
short-circuit operators 89
side effect 82
simple stream editor 237
`sin` 134
single character fields 46
single quotes, why needed 10
skipping input files 167
skipping lines between markers 98
sparse arrays 122
`split` 137
`split` utility 212
`split.awk` 213
`sprintf` 137
`sqrt` 134
`srand` 135
Stallman, Richard 1, 3
standard error output 69
standard input 11, 37, 69
standard output 69
statement, compound 103
stream editor 48
stream editor, simple 237
`strftime` 147
string comparison vs. regexp comparison 88
string constants 75
string operators 81
string-matching operators 23
`sub` 138
`sub`, third argument of 139
subscripts in arrays 128
`SUBSEP` 114, 128
`substr` 140
subtraction 80
`system` 144
`systime` 146

T

Tcl 200
`tee` utility 214
`tee.awk` 214
terminator, record 39
time of day 146
timestamps 146
timestamps, converting from dates ... 175
timestamps, formatted 180
`tolower` 141
`toupper` 141
`translate.awk` 227
Trueman, David 3
truth values 86
type conversion 79
types of variables 82, 86

U

undefined functions 155
undocumented features 165
uninitialized variables, as array subscripts 127
`uniq` utility 216
`uniq.awk` 217
use of comments 13
user information 189
user-defined functions 151
user-defined variables 77
uses of `awk` 5
using this book 5

V

values of characters as numbers 172
variable shadowing 152
variable typing 86
variables, user-defined 77

W

Wall, Larry 293
`wc` utility 220
`wc.awk` 221
Weinberger, Peter 1
when to use `awk` 18
`while` statement 104
word boundaries, matching 32
word, regexp definition of 31
`wordfreq.sh` 231

Short Contents

Preface 1
1 Introduction 5
2 Getting Started with `awk` 9
3 Useful One Line Programs 21
4 Regular Expressions 23
5 Reading Input Files 37
6 Printing Output 59
7 Expressions 75
8 Patterns and Actions 95
9 Control Statements in Actions 103
10 Built-in Variables 113
11 Arrays in `awk` 121
12 Built-in Functions 133
13 User-defined Functions 151
14 Running `awk` 159
15 A Library of `awk` Functions 167
16 Practical `awk` Programs 201
17 The Evolution of the `awk` Language 247
Appendix A `gawk` Summary 253
Appendix B Installing `gawk` 273
Appendix C Implementation Notes 289
Appendix D Glossary 295
GNU GENERAL PUBLIC LICENSE 303
Index 311

Table of Contents

Preface 1
History of awk and gawk 1
The GNU Project and This Book 1
Acknowledgements 3

1 Introduction 5
1.1 Using This Book 5
Dark Corners 6
1.2 Typographical Conventions 6
1.3 Data Files for the Examples 7

2 Getting Started with awk 9
2.1 A Rose By Any Other Name 9
2.2 How to Run awk Programs 10
2.2.1 One-shot Throw-away awk Programs 10
2.2.2 Running awk without Input Files 11
2.2.3 Running Long Programs 11
2.2.4 Executable awk Programs 12
2.2.5 Comments in awk Programs 13
2.3 A Very Simple Example 13
2.4 An Example with Two Rules 14
2.5 A More Complex Example 15
2.6 awk Statements Versus Lines 16
2.7 Other Features of awk 18
2.8 When to Use awk 18

3 Useful One Line Programs 21

4 Regular Expressions 23
4.1 How to Use Regular Expressions 23
4.2 Escape Sequences 24
4.3 Regular Expression Operators 26
4.4 Additional Regexp Operators Only in gawk 31
4.5 Case-sensitivity in Matching 33
4.6 How Much Text Matches? 34
4.7 Using Dynamic Regexps 35

5 Reading Input Files 37

5.1 How Input is Split into Records 37
5.2 Examining Fields 40
5.3 Non-constant Field Numbers 41
5.4 Changing the Contents of a Field 42
5.5 Specifying How Fields are Separated 44
5.5.1 The Basics of Field Separating 44
5.5.2 Using Regular Expressions to Separate Fields 45
5.5.3 Making Each Character a Separate Field 46
5.5.4 Setting `FS` from the Command Line 46
5.5.5 Field Splitting Summary 48
5.6 Reading Fixed-width Data 49
5.7 Multiple-Line Records 50
5.8 Explicit Input with `getline` 53
5.8.1 Introduction to `getline` 53
5.8.2 Using `getline` with No Arguments 53
5.8.3 Using `getline` Into a Variable 54
5.8.4 Using `getline` from a File 55
5.8.5 Using `getline` Into a Variable from a File 56
5.8.6 Using `getline` from a Pipe 56
5.8.7 Using `getline` Into a Variable from a Pipe 57
5.8.8 Summary of `getline` Variants 58

6 Printing Output 59

6.1 The `print` Statement 59
6.2 Examples of `print` Statements 59
6.3 Output Separators 61
6.4 Controlling Numeric Output with `print` 62
6.5 Using `printf` Statements for Fancier Printing 62
6.5.1 Introduction to the `printf` Statement 62
6.5.2 Format-Control Letters 63
6.5.3 Modifiers for `printf` Formats 64
6.5.4 Examples Using `printf` 66
6.6 Redirecting Output of `print` and `printf` 67
6.7 Special File Names in `gawk` 69
6.8 Closing Input and Output Files and Pipes 71

7 Expressions 75

7.1 Constant Expressions 75
7.1.1 Numeric and String Constants 75
7.1.2 Regular Expression Constants 75
7.2 Using Regular Expression Constants 76
7.3 Variables 77
7.3.1 Using Variables in a Program 77
7.3.2 Assigning Variables on the Command Line 78
7.4 Conversion of Strings and Numbers 79
7.5 Arithmetic Operators 80
7.6 String Concatenation 81
7.7 Assignment Expressions 82
7.8 Increment and Decrement Operators 85
7.9 True and False in `awk` 86
7.10 Variable Typing and Comparison Expressions 86
7.11 Boolean Expressions 89
7.12 Conditional Expressions 90
7.13 Function Calls 91
7.14 Operator Precedence (How Operators Nest) 92

8 Patterns and Actions 95

8.1 Pattern Elements 95
8.1.1 Kinds of Patterns 95
8.1.2 Regular Expressions as Patterns 95
8.1.3 Expressions as Patterns 96
8.1.4 Specifying Record Ranges with Patterns 97
8.1.5 The `BEGIN` and `END` Special Patterns 98
8.1.5.1 Startup and Cleanup Actions 98
8.1.5.2 Input/Output from `BEGIN` and `END` Rules 99
8.1.6 The Empty Pattern 100
8.2 Overview of Actions 100

9 Control Statements in Actions 103

9.1 The `if-else` Statement 103
9.2 The `while` Statement 104
9.3 The `do-while` Statement 104
9.4 The `for` Statement 105
9.5 The `break` Statement 106
9.6 The `continue` Statement 107
9.7 The `next` Statement 108
9.8 The `nextfile` Statement 109
9.9 The `exit` Statement 110

10 Built-in Variables 113

10.1 Built-in Variables that Control `awk` 113
10.2 Built-in Variables that Convey Information 115
10.3 Using `ARGC` and `ARGV` 117

11 Arrays in `awk` 121

11.1 Introduction to Arrays 121
11.2 Referring to an Array Element 122
11.3 Assigning Array Elements 123
11.4 Basic Array Example 123
11.5 Scanning All Elements of an Array 124
11.6 The `delete` Statement 125
11.7 Using Numbers to Subscript Arrays 127
11.8 Using Uninitialized Variables as Subscripts 127
11.9 Multi-dimensional Arrays 128
11.10 Scanning Multi-dimensional Arrays 130
11.11 Using Array Memory Efficiently 130

12 Built-in Functions 133

12.1 Calling Built-in Functions 133
12.2 Numeric Built-in Functions 134
12.3 Built-in Functions for String Manipulation 135
12.4 Built-in Functions for Input/Output 143
12.5 Functions for Dealing with Time Stamps 146

13 User-defined Functions 151

13.1 Function Definition Syntax 151
13.2 Function Definition Examples 152
13.3 Calling User-defined Functions 154
13.4 The `return` Statement 156

14 Running `awk` 159

14.1 Command Line Options 159
14.2 Other Command Line Arguments 163
14.3 The `AWKPATH` Environment Variable 164
14.4 Obsolete Options and/or Features 165
14.5 Undocumented Options and Features 165
14.6 Known Bugs in `gawk` 165

15 A Library of awk Functions 167
15.1 Simulating gawk-specific Features 167
15.2 Implementing nextfile as a Function 167
15.3 Assertions 169
15.4 Rounding Numbers 171
15.5 Translating Between Characters and Numbers 172
15.6 Merging an Array Into a String 174
15.7 Turning Dates Into Timestamps 175
15.8 Managing the Time of Day 180
15.9 Noting Data File Boundaries 182
15.10 Processing Command Line Options 184
15.11 Reading the User Database 189
15.12 Reading the Group Database 194
15.13 Naming Library Function Global Variables 199

16 Practical awk Programs 201
16.1 Re-inventing Wheels for Fun and Profit 201
16.1.1 Cutting Out Fields and Columns 201
16.1.2 Searching for Regular Expressions in Files 206
16.1.3 Printing Out User Information 210
16.1.4 Splitting a Large File Into Pieces 212
16.1.5 Duplicating Output Into Multiple Files 214
16.1.6 Printing Non-duplicated Lines of Text 216
16.1.7 Counting Things 220
16.2 A Grab Bag of awk Programs 223
16.2.1 Finding Duplicated Words in a Document 223
16.2.2 An Alarm Clock Program 223
16.2.3 Transliterating Characters 226
16.2.4 Printing Mailing Labels 228
16.2.5 Generating Word Usage Counts 230
16.2.6 Removing Duplicates from Unsorted Text 232
16.2.7 Extracting Programs from Texinfo Source Files 233
16.2.8 A Simple Stream Editor 237
16.2.9 An Easy Way to Use Library Functions 239

17 The Evolution of the awk Language 247
17.1 Major Changes between V7 and SVR3.1 247
17.2 Changes between SVR3.1 and SVR4 248
17.3 Changes between SVR4 and POSIX awk 248
17.4 Extensions in the Bell Laboratories awk 249
17.5 Extensions in gawk Not in POSIX awk 249

Appendix A gawk Summary 253
A.1 Command Line Options Summary 253
A.2 Language Summary 254
A.3 Variables and Fields 255
A.3.1 Fields 255
A.3.2 Built-in Variables 256
A.3.3 Arrays 258
A.3.4 Data Types 258
A.4 Patterns 259
A.4.1 Pattern Summary 259
A.4.2 Regular Expressions 260
A.5 Actions 262
A.5.1 Operators 262
A.5.2 Control Statements 263
A.5.3 I/O Statements 264
A.5.4 `printf` Summary 265
A.5.5 Special File Names 267
A.5.6 Built-in Functions 267
A.5.7 Time Functions 269
A.5.8 String Constants 270
A.6 User-defined Functions 271
A.7 Historical Features 271

Appendix B Installing gawk 273
B.1 The `gawk` Distribution 273
B.1.1 Getting the `gawk` Distribution 273
B.1.2 Extracting the Distribution 275
B.1.3 Contents of the `gawk` Distribution 275
B.2 Compiling and Installing `gawk` on Unix 278
B.2.1 Compiling `gawk` for Unix 278
B.2.2 The Configuration Process 279
B.3 How to Compile and Install `gawk` on VMS 279
B.3.1 Compiling `gawk` on VMS 279
B.3.2 Installing `gawk` on VMS 280
B.3.3 Running `gawk` on VMS 281
B.3.4 Building and Using `gawk` on VMS POSIX 281
B.4 MS-DOS and OS/2 Installation and Compilation 282
B.5 Installing `gawk` on the Atari ST 283
B.5.1 Compiling `gawk` on the Atari ST 283
B.5.2 Running `gawk` on the Atari ST 284
B.6 Installing `gawk` on an Amiga 285
B.7 Reporting Problems and Bugs 285
B.8 Other Freely Available `awk` Implementations 287

Appendix C Implementation Notes 289
C.1 Downward Compatibility and Debugging 289
C.2 Making Additions to `gawk` 289
C.2.1 Adding New Features 289
C.2.2 Porting `gawk` to a New Operating System 291
C.3 Probable Future Extensions 293
C.4 Suggestions for Improvements 294

Appendix D Glossary 295

GNU GENERAL PUBLIC LICENSE 303
Preamble 303
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION 304
How to Apply These Terms to Your New Programs 309

Index 311