*In this chapter:*

- *Running the Example Programs*
- *Reinventing Wheels for Fun and Profit*
- *A Grab Bag of awk Programs*

# 13

# *Practical awk Programs*

Chapter 12, *A Library of awk Functions*, presents the idea that reading programs in a language contributes to learning that language. This chapter continues that theme, presenting a potpourri of *awk* programs for your reading enjoyment. There are three sections. The first describes how to run the programs presented in this chapter.

The second presents *awk* versions of several common POSIX utilities. These are programs that you are hopefully already familiar with, and therefore, whose problems are understood. By reimplementing these programs in *awk*, you can focus on the *awk*-related aspects of solving the programming problem.

The third is a grab bag of interesting programs. These solve a number of different data-manipulation and management problems. Many of the programs are short, which emphasizes *awk*'s ability to do a lot in just a few lines of code.

Many of these programs use the library functions presented in Chapter 12.

## *Running the Example Programs*

To run a given program, you would typically do something like this:

```
awk -f program -- options files
```

Here, *program* is the name of the *awk* program (such as *cut.awk*), *options* are any command-line options for the program that start with a –, and *files* are the actual datafiles.

If your system supports the `#!` executable interpreter mechanism (see the section "Executable awk Programs" in Chapter 1, *Getting Started with awk*), you can instead run your program directly:

```
cut.awk -c1-8 myfiles > results
```

If your *awk* is not *gawk*, you may instead need to use this:

```
cut.awk -- -c1-8 myfiles > results
```

# Reinventing Wheels for Fun and Profit

This section presents a number of POSIX utilities that are implemented in *awk*. Reinventing these programs in *awk* is often enjoyable, because the algorithms can be very clearly expressed, and the code is usually 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.

## Cutting out Fields and Columns

The *cut* utility selects, or "cuts," characters or fields from its standard input and sends them to its standard output. Fields are separated by tabs by default, 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 1 through 8, 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 the section "Processing Command-Line Options" in Chapter 12) and the `join` library function (see the section "Merging an Array into a String" in Chapter 12).

The program begins with a comment describing the options, the library functions needed, and a `usage` function that prints out a usage message and exits. `usage` is called if invalid arguments are supplied:

```
# cut.awk --- implement cut in awk

# Options:
#    -f list     Cut fields
#    -d c        Field delimiter character
#    -c list     Cut characters
#
#    -s          Suppress lines without the delimiter
#
# Requires getopt and join library functions

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, because 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 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 1 to `Optind`, so that *awk* does not try to process the command-line options as filenames.

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 the program calls either `set_fieldlist` or `set_charlist` 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()
}
```

`set_fieldlist` is used to split the field list apart at the commas and 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 the section "Reading Fixed-Width Data" in Chapter 3, *Reading Input Files*), 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 print and also the intervening characters that have to be skipped. For example, suppose you wanted characters 1 through 8, 15, and 22 through 35. You would use `-c 1-8,15,22-35`. The necessary value for `FIELDWIDTHS` is `"8 6 1 6 14"`. This yields five fields, and the fields to print are `$1`, `$3`, and `$5`. The intermediate fields are *filler*, which is stuff in between the desired data. `flist` lists the fields to print, 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
}
```

Next is the rule that actually processes the data. If the *–s* option is given, then `suppress` is 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 *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. The corresponding field is printed if it contains data. If the next field also has data, then the separator character is written out between the fields:

```
{
    if (by_fields && suppress && index($0, FS) != 0)
        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 is possible in other *awk* implementations to use `substr` (see the section "String-Manipulation Functions" in Chapter 8, *Functions*), it is also extremely painful. The `FIELDWIDTHS` variable supplies an elegant solution to the problem of picking the input line apart by characters.

## *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 Chapter 2, *Regular Expressions*). It is used in the following manner:

```
egrep [ options ] 'pattern' files ...
```

The *pattern* is a regular expression. In typical usage, the regular expression is quoted to prevent the shell from expanding any of the special characters as filename wildcards. Normally, *egrep* prints the lines that matched. If multiple filenames are provided on the command line, each output line is preceded by the name of the file and a colon.

The options to *egrep* are as follows:

`-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 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 is not matched.

`-i` Ignore case distinctions in both the pattern and the input data.

`-l` Only print (list) 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 the section "Processing Command-Line Options" in Chapter 12) and the file transition library program (see the section "Noting Datafile Boundaries" in Chapter 12).

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 the section "Built-in Variables" in Chapter 6, *Patterns, Actions, and Variables*):

```
# egrep.awk --- simulate egrep in awk

# 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
#
# Requires getopt and file transition library functions

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 is supplied with *–e*, the first nonoption 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 are specified, the standard input is used, and if multiple files are specified, we make sure to note this so that the filenames can precede the matched lines in the output:

```
    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 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*.

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 lowercase if the *–i* option is 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 (naming the parameter `junk` shows we know that `beginfile` is called with a parameter, but that we're not interested in its value):

```
function beginfile(junk)
{
    fcount = 0
}
```

The `endfile` function is called after each file has been processed. It affects the output only when the user wants a count of the number of lines that matched. `no_print` is true only if the exit status is desired. `count_only` is true if line counts are desired. *egrep* therefore only prints line counts if printing and counting are enabled. The output format must be adjusted depending upon the number of files to process. Finally, `fcount` is added to `total`, so that we know the total number of lines that matched the pattern:

```
function endfile(file)
{
    if (! no_print && count_only)
        if (do_filenames)
            print file ":" fcount
        else
            print fcount
```

* It also introduces a subtle bug; if a match happens, we output the translated line, not the original.

```
        total += fcount
    }
```

The following rule does most of the work of matching lines. The variable `matches` is true if the line matched the pattern. If the user wants lines that did not match, the sense of `matches` is inverted using the `!` operator. `fcount` is incremented with the value of `matches`, which is either one or zero, depending upon a successful or unsuccessful match. If the line does not match, the `next` statement just moves on to the next record.

A number of additional tests are made, but they are only done if we are not counting lines. First, if the user only wants exit status (`no_print` is true), then it is enough to know that *one* line in this file matched, and we can skip on to the next file with `nextfile`. Similarly, if we are only printing filenames, we can print the filename, 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 (! count_only) {
        if (no_print)
            nextfile

        if (filenames_only) {
            print FILENAME
            nextfile
        }

        if (do_filenames)
            print FILENAME ":" $0
        else
            print
    }
}
```

The `END` rule takes care of producing the correct exit status. If there are 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 ...]"
    e = e "\n\tegrep [-csvil] 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.

## *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* only prints the effective user ID and group ID only if they are different from the real ones. If possible, *id* also supplies 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 part of what is provided by *gawk*'s `PROCINFO` array (see the section "Built-in Variables" in Chapter 6). However, the *id* utility provides a more palatable output than just individual numbers.

Here is a simple version of *id* written in *awk*. It uses the user database library functions (see the section "Reading the User Database" in Chapter 12) and the group database library functions (see the section "Reading the Group Database" in Chapter 12).

The program is fairly straightforward. All the work is done in the `BEGIN` rule. The user and group ID numbers are obtained from `PROCINFO`. 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
#
# Requires user and group library functions
# output is:
# uid=12(foo) euid=34(bar) gid=3(baz) \
#             egid=5(blat) groups=9(nine),2(two),1(one)

BEGIN    \
{
    uid = PROCINFO["uid"]
    euid = PROCINFO["euid"]
    gid = PROCINFO["gid"]
    egid = PROCINFO["egid"]

    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])
        }
    }
```

```
        for (i = 1; ("group" i) in PROCINFO; i++) {
            if (i == 1)
                printf(" groups=")
            group = PROCINFO["group" i]
            printf("%d", group)
            pw = getgrgid(group)
            if (pw != "") {
                split(pw, a, ":")
                printf("(%s)", a[1])
            }
            if (("group" (i+1)) in PROCINFO)
                printf(",")
        }

        print ""
    }
```

The test in the `for` loop is worth noting. Any supplementary groups in the `PROCINFO` array have the indices `"group1"` through `"group`*N*`"` for some *N*, i.e., the total number of supplementary groups. However, we don't know in advance how many of these groups there are.

This loop works by starting at one, concatenating the value with `"group"`, and then using `in` to see if that value is in the array. Eventually, `i` is incremented past the last group in the array and the loop exits.

The loop is also correct if there are *no* supplementary groups; then the condition is false the first time it's tested, and the loop body never executes.

## *Splitting a Large File into Pieces*

The `split` program splits large text files into smaller pieces. Usage is as follows:

```
split [-count] file [ prefix ]
```

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, 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, supply an additional argument that specifies the filename prefix.

Here is a version of `split` in *awk*. It uses the `ord` and `chr` functions presented in the section "Translating Between Characters and Numbers" in Chapter 12.

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 sign 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 filename is skipped over and the final argument is used as the prefix for the output filenames:

```
# split.awk --- do split in awk
#
# Requires ord and chr library functions
# 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 filename. 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 automatically close the last file instead of doing it in an `END` rule. It also assumes that letters are contiguous in the character set, which isn't true for EBCDIC systems.

## *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 as follows:

```
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* attempts to process each filename 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 filenames 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

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 is also possible to write 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 $M$ output files, the first method only executes $N$ `if` statements, while the second executes $N \times M$ `if` statements.

Finally, the `END` rule cleans up by closing all the output files:

```
END    \
{
    for (i in copy)
        close(copy[i])
}
```

## Printing Nonduplicated Lines of Text

The *uniq* utility reads sorted lines of data on its standard input, and by default removes duplicate lines. In other words, it only prints unique lines—hence the name. *uniq* has a number of options. The usage is as follows:

```
uniq [-udc [-n]] [+n] [ input file [ output file ]]
```

The options for *uniq* are:

`-d` Print only repeated lines.

`-u` Print only nonrepeated lines.

`-c` Count lines. This option overrides *–d* and *–u*. Both repeated and nonrepeated lines are counted.

`-`*n* Skip *n* fields before comparing lines. The definition of fields is similar to *awk*'s default: nonwhitespace 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 are provided.

*uniq* uses the `getopt` library function (see the section "Processing Command-Line Options" in Chapter 12) and the `join` library function (see the section "Merging an Array into a String" in Chapter 12).

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 are supplied (`Optarg` looks like a number), `Optarg` is concatenated with the option digit and then the 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. In this case, `Optind` must be decremented so that `getopt` processes it next time. This code is admittedly a bit tricky.

If no options are supplied, then the default is taken, to print both repeated and nonrepeated lines. The output file, if provided, is assigned to `outputfile`. Early on, `outputfile` is initialized to the standard output, */dev/stdout*:

```
# uniq.awk --- do uniq in awk
#
# Requires getopt and join library functions
```

```
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 are 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 the section "String-Manipulation Functions" in Chapter 8); the desired fields are then 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` is one or zero, depending upon the results of `are_equal`'s comparison. If *uniq* is counting repeated lines, and the lines are equal, then it increments the `count` variable. Otherwise, it prints the line and resets `count`, since the two lines are not equal.

If *uniq* is not counting, and if the lines are equal, `count` is incremented. Nothing is printed, since the point is to remove duplicates. Otherwise, if *uniq* is counting repeated lines and more than one line is seen, or if *uniq* is counting nonrepeated lines and only one line is 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
}
```

## Counting Things

The *wc* (word count) utility counts lines, words, and characters in one or more input files. Its usage is as follows:

```
wc [-lwc] [ files ... ]
```

If no files are specified on the command line, *wc* reads its standard input. If there are multiple files, it also prints total counts for all the files. The options and their meanings are shown in the following list:

`-l` Count only lines.

`-w` Count only words. A "word" is a contiguous sequence of nonwhitespace characters, separated by spaces and/or tabs. Luckily, this is the normal way *awk* separates fields in its input data.

`-c` Count only 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), and it can easily tell us how long a line is.

This uses the `getopt` library function (see the section "Processing Command-Line Options" in Chapter 12) and the file-transition functions (see the section "Noting Datafile Boundaries" in Chapter 12).

This version has one notable difference from traditional versions of *wc*: it 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` is true if more than one file is named on the command line:

```
# wc.awk --- count lines, words, characters

# Options:
#    -l    only count lines
#    -w    only count words
#    -c    only count characters
#
# Default is to count lines, words, characters
#
# Requires getopt and file transition library functions

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 filename in `fname`:

```
function beginfile(file)
{
    chars = lines = words = 0
    fname = FILENAME
}
```

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 datafile:

```
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, plus one, to `chars`. Adding one plus the record length is needed because the newline character separating records (the value of `RS`) is not part of the record itself, and thus not included in its length. Next, `lines` is incremented for each line read, and `words` is incremented by the value of `NF`, which is the number of "words" on this line:

```
# do per line
{
    chars += length($0) + 1    # get newline
    lines++
    words += NF
}
```

* *wc* can't just use the value of `FNR` in `endfile`. If you examine the code in the section "Noting Datafile Boundaries" in Chapter 12, you will see that `FNR` has already been reset by the time `endfile` is called.

Finally, the `END` rule simply prints the totals for all the files:

```
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"
    }
}
```

# *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.

## *Finding Duplicated Words in a Document*

A common error when writing large amounts of prose is to accidentally duplicate words. Typically you will see this in text as something like "the program does the following..." When the text is online, 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 lowercase, so that, for example, "The" and "the" compare equal to each other. The next statement replaces nonalphanumeric and nonwhitespace characters with spaces, so that punctuation does not affect the comparison either. The characters are replaced with spaces so that formatting controls don't create nonsense words (e.g., the Texinfo `@code{NF}` becomes `codeNF` if punctuation is simply deleted). The record is then resplit into fields, yielding just the actual words on the line, and ensuring that there are no empty fields.

If there are no fields left after removing all the punctuation, the current record is skipped. Otherwise, the program loops through each word, comparing it to the previous one:

```
# dupword.awk --- find duplicate words in text
```

```
{
    $0 = tolower($0)
    gsub(/[^[:alnum:][:blank:]]/, " ");
    $0 = $0         # re-split
    if (NF == 0)
        next
    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
}
```

## 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 specified time, it prints the message on the standard output. In addition, you can give it the number of times to repeat the message as well as a delay between repetitions.

This program uses the `gettimeofday` function from the section "Managing the Time of Day" in Chapter 12.

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 without 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 an audible alert. Thus when the alarm goes off, the system calls attention to itself in case the user is not looking at the computer or terminal.) Here is the program:

```
# alarm.awk --- set an alarm
#
# Requires gettimeofday library function

# 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 usage1 > "/dev/stderr"
    print usage2 > "/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, converts it (if necessary) to a 24-hour clock, and then 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 alarm 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 the section "Input/Output Functions" in Chapter 8) 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
    }
```

## *Transliterating Characters*

The system *tr* utility transliterates characters. For example, it is often used to map uppercase letters into lowercase for further processing:

```
generate data | tr 'A-Z' 'a-z' | process data ...
```

*tr* requires two lists of characters.* 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.

* On some older System V systems, including Solaris, *tr* may require that the lists be written as range expressions enclosed in square brackets (`[a-z]`) and quoted, to prevent the shell from attempting a filename expansion. This is not a feature.

Some time ago, a user proposed that a transliteration function should be added to *gawk*. The following program was written 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 does 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 the section "String-Manipulation Functions" in Chapter 8).* There are two functions. The first, `stranslate`, takes three arguments:

`from`
: A list of characters from which to translate.

`to`
: A list of characters from which to translate.

`target`
: The string on which to do the translation

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* reads from the standard input.

Finally, the processing rule simply calls `translate` for each record:

```
# translate.awk --- do tr-like stuff

# 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)
```

* This program was written before *gawk* acquired the ability to split each character in a string into separate array elements.

```
        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 (the *gawk* authors) 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 (see the section "String-Manipulation Functions" in Chapter 8). 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.

## *Printing Mailing Labels*

This next script reads lists of names and addresses and generates mailing labels. Each page of labels has 20 labels on it, 2 across and 10 down. The addresses are guaranteed to be no more than 5 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* splits records at blank lines (see the section "How Input Is Split into Records" in Chapter 3). It sets `MAXLINES` to 100, since 100 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 print 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 0 to 4, `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, an extra blank line is printed at lines 21 and 61, 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 2 blank lines at the top and 2 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 --- print mailing 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()
    }
```

## *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` *`index`* `in` *`array`* mechanism. Finally, it shows how *awk* is 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:

```
# 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]
}
```

This program has two rules. The first rule, because it has an empty pattern, is executed for every input line. It uses *awk*'s field-accessing mechanism (see the section "Examining Fields" in Chapter 3) to pick out the individual words from the line, and the built-in variable `NF` (see the section "Built-in Variables" in Chapter 6) to know how many fields are available. For each input word, it increments an element of the array `freq` 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 just by whitespace. 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 lowercase 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 in having an alphabetized table of how frequently each word occurs.

The way to solve these problems is to use some of *awk*'s more advanced features. 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:

```
# wordfreq.awk --- print list of word frequencies

{
    $0 = tolower($0)    # remove case distinctions
    gsub(/[^[:alnum:]_[:blank:]]/, "", $0)  # remove punctuation
    for (i = 1; i <= NF; i++)
        freq[$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* specify a sort that uses 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.

The *sort* could even be done 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)
}
```

This way of sorting must be used on systems that do not have true pipes at the command-line (or batch-file) level. See the general operating system documentation for more information on how to use the *sort* program.

## *Removing Duplicates from Unsorted Text*

The *uniq* program (see the section "Printing Nonduplicated Lines of Text" earlier in this chapter) removes duplicate lines from *sorted* data.

Suppose, however, you need to remove duplicate lines from a datafile but that you want 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 want 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]` is zero. In this 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 are encountered. The `END` rule simply prints out the lines, in order:

```
# histsort.awk --- compact a shell history file
# 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` statement in the `END` rule indicates how often a particular command is used:

```
print data[lines[i]], lines[i]
```

This works because `data[$0]` is incremented each time a line is seen.

## *Extracting Programs from Texinfo Source Files*

Both this chapter and Chapter 12 present a large number of *awk* programs. If you want 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 online documentation. Texinfo is fully documented in the book *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 as the backslash (\) is in C or *awk*. Literal `@` symbols are represented in Texinfo source files as `@@`.
- Comments start with either `@c` or `@comment`. The file-extraction program works by using special comments that start at the beginning of a line.
- Lines containing `@group` and `@end group` commands bracket example text that should not be split across a page boundary. (Unfortunately, TEX isn't always smart enough to do things exactly right, and we have to give it some help.)

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

* The book was translated into DocBook XML for the O'Reilly & Associates edition.

to the `system` function (see the section "Input/Output Functions" in Chapter 8). Upon seeing `@c file` *`filename`*, each subsequent line is sent to the file *filename*, until `@c endfile` is encountered. The rules in *extract.awk* 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 the section "Merging an Array into a String" in Chapter 12).

The example programs in the online 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} rule,
that 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- and lowercase letters in the directives won't matter.

The first rule handles calling `system`, checking that a command is given (`NF` is at least three) and also checking that the command exits with a zero-exit status, signifying OK:

```
# extract.awk --- extract files and run programs
#                 from texinfo files

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 filename is given in the directive. If the file named is not the current file, then the current file is closed. Keeping the current file open until a new file is encountered allows the use of the `>` redirection for printing the contents, keeping open file management simple.

The `for` loop does the work. It reads lines using `getline` (see the section "Explicit Input with getline" in Chapter 3). 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. Similarly, comments within examples are also ignored.

Most of the work is in the following few lines. If the line has no `@` symbols, the program can print it 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 the section "String-Manipulation Functions" in Chapter 8). The `@` symbol is used as the separator character. 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 a single `@` symbol back in.

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
            else if (line ~ /^@c(omment+)?[ \t]+/)
                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 the section "Redirecting Output of print and printf" in Chapter 4, *Printing Output*). This makes it easy to 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 filename 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)
}
```

## 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 it 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, `s/old/new/g` tells *sed* to look for the regexp `old` on each input line and globally replace it with the text `new`, i.e., all the occurrences on a line. This is similar to *awk*'s `gsub` function (see the section "String-Manipulation Functions" in Chapter 8).

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 filenames 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

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, as well as on the setting of `RT` to the actual text that terminates the record (see the section "How Input Is Split into Records" in Chapter 3).

The idea is to have RS be the pattern to look for. *gawk* automatically sets $0 to the text between matches of the pattern. This is text that we want to keep, unmodified. Then, by setting ORS to the replacement text, a simple print statement outputs the text we want 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 is set to the null string. In this case, we can print $0 using printf (see the section "Using printf Statements for Fancier Printing" in Chapter 4).

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 are not treated as filenames (see the section "Using ARGC and ARGV" in Chapter 6).

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.

## An Easy Way to Use Library Functions

Using library functions in *awk* can be very beneficial. It encourages code reuse 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 the section "Command-Line Options" in Chapter 11, *Running awk and gawk*). It would be nice to be able to write programs in the following manner:

```
# 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 is included with `@include` can contain further `@include` statements. *igawk* makes an effort to only include files once, so that nested includes don't accidentally include a library function twice.

*igawk* should behave just like *gawk* externally. 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. It works 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 automatically supplies a trailing newline.
   b. Source filenames, provided with *–f*. We use a neat trick and echo `@include` *`filename`* into the temporary file. Since the file-inclusion program works the way *gawk* does, this gets 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 filenames).

The initial part of the program turns on shell tracing if the first argument is `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 filename 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, runs `gawk –version` to get the *gawk* version information, and then exits.

If none of the *–f*, *––file*, *–Wfile*, *––source*, or *–Wsource* arguments are supplied, then the first nonoption 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 filenames. This allows multiple users to run *igawk* without worrying that the temporary filenames will clash. The program is as follows:

```
#! /bin/sh
# igawk --- like gawk but do @include processing

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 the section "Explicit Input with getline" in Chapter 3). The input filenames and `@include` statements are managed using a stack. As each `@include` is encountered, the current filename is "pushed" onto the stack and

the file named in the `@include` directive becomes the current filename. 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 the section "The AWKPATH Environment Variable" in Chapter 11). If a filename has a `/` in it, no path search is done. Otherwise, the filename is concatenated with the name of each directory in the path, and an attempt is made to open the generated filename. The only way to test if a file can be read in *awk* is to go ahead and try to read it with `getline`; this is what `pathto` does.* If the file can be read, it is closed and the filename 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] = "."
    }
```

* On some very old versions of *awk*, the test `getline junk < t` can loop forever if the file exists but is empty. Caveat emptor.

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. If the line does start with `@include`, the filename is in `$2`. `pathto` is called to generate the full path. If it cannot, then we print an error message and continue.

The next thing to check is if the file is included already. The `processed` array is indexed by the full filename of each included file and it tracks this information for us. If the file is seen again, a warning message is printed. Otherwise, the new filename 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  # push onto stack
                } 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, along with 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 make the program work better:

- 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.
- Not trying to save the line read with `getline` when testing for the file's accessibility for use with the main program complicates things considerably.
- 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 contains a set of default library functions, such as `getopt` and `assert`.

*site.awk*
: This file contains 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.