1 Getting Started with awk

In this chapter:

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 it reaches the end of the input files.

Programs in *awk* are different from programs in most other languages, because *awk* programs are *data-driven*; that is, you describe the data you want 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 read and write.

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 that we will ignore for now. See the section "User-Defined Functions" in Chapter 8, *Functions*.) Each rule specifies one pattern to search for and one action to perform upon finding the pattern.

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

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

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 ...
```

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 ...
```

This section discusses both mechanisms, along with several variations of each.

One-Shot Throwaway 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 the shell won't interpret any *awk* characters as special shell characters. The quotes 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 because there are no other files to misplace.

The section "Some Simple Examples" later in this chapter presents several short, self-contained programs.

Running awk Without Input Files

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

```
awk 'program'
```

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 Ctrl-d. (On other operating systems, the end-of-file character may be different. For example, on OS/2 and MS-DOS, it is Ctrl-z.)

As an example, the following program prints a friendly piece of advice (from Douglas Adams's *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 on 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?
Ctrl-d
```

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. In order 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 filename 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!\" }"
```

* Although we generally recommend the use of single quotes around the program text, double quotes are needed here in order to put the single quote into the message.

This was explained earlier (see the previous section “Running awk Without Input Files).” Note that you don’t usually need single quotes around the filename that you specify with *–f*, because most filenames 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 filename. This doesn’t affect the execution of the *awk* program but it does make “housekeeping” easier.

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[*] as well as 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), simply type `advice` at the shell and the system arranges to run *awk*[†] as if you had typed `awk -f advice`:

```
$ chmod +x advice
$ advice
Don’t Panic!
```

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

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 an executable part of the program. Comments can explain what the program does and how it works. Nearly all programming languages have provisions for comments, as programs are typically hard to understand without them.

[*] The `#!` mechanism works on Linux systems, systems derived from the 4.4-Lite Berkeley Software Distribution, and most commercial Unix systems.

[†] The line beginning with `#!` lists the full filename of an interpreter to 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 filename of the *awk* program. The rest of the argument list contains either options to *awk*, or datafiles, or both.

Portability Issues with #!

Some systems limit the length of the interpreter name to 32 characters. Often, this can be dealt with by using a symbolic link.

You should not put more than one argument on the `#!` line after the path to *awk*. It does not work. The operating system treats the rest of the line as a single argument and passes it to *awk*. Doing this leads to confusing behavior—most likely a usage diagnostic of some sort from *awk*.

Finally, the value of `ARGV[0]` (see the section "Built-in Variables" in Chapter 6, *Patterns, Actions, and Variables*) varies depending upon your operating system. Some systems put `awk` there, some put the full pathname of *awk* (such as */bin/awk*), and some put the name of your script (`advice`). Don't rely on the value of `ARGV[0]` to provide your script name.

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 throwaway *awk* programs, but this usually isn't very useful; the purpose of a comment is to help you or another person understand the program when reading it at a later time.

Shell-Quoting Issues

For short to medium length *awk* programs, it is most convenient to enter the program on the *awk* command line. This is best done by enclosing the entire program in single quotes. This is true whether you are entering the program interactively at the shell prompt or writing it as part of a larger shell script:

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

Once you are working with the shell, it is helpful to have a basic knowledge of shell-quoting rules. The following rules apply only to POSIX-compliant, Bourne-style shells (such as *bash*, the GNU Bourne-again shell). If you use *csh*, you're on your own:

As mentioned in the section "One-Shot Throwaway awk Programs" earlier in this chapter, you can enclose small- to medium-sized 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 interprets the quote as the closing quote for the entire program. As a result, usually the shell prints a message about mismatched quotes, and if *awk* actually runs, it will probably print strange messages about syntax errors. For example, look at the following:

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

The shell sees that the first two quotes match, and that a new quoted object begins at the end of the command line. It therefore prompts with the secondary prompt, waiting for more input. With Unix *awk*, closing the quoted string produces this result:

```
$ awk '{ print "hello" } # let's be cute'
> '
awk: can't open file be
 source line number 1
```

Putting a backslash before the single quote in `let's` wouldn't help, since backslashes are not special inside single quotes. The next section describes the shell's quoting rules.

- Quoted items can be concatenated with nonquoted items as well as with other quoted items. The shell turns everything into one argument for the command.
- Preceding any single character with a backslash (`\`) quotes that character. The shell removes the backslash and passes the quoted character on to the command.
- Single quotes protect everything between the opening and closing quotes. The shell does no interpretation of the quoted text, passing it on verbatim to the command. It is *impossible* to embed a single quote inside single-quoted text. Refer back to the section "Comments in awk Programs" earlier in this chapter for an example of what happens if you try.
- Double quotes protect most things between the opening and closing quotes. The shell does at least variable and command substitution on the quoted text. Different shells may do additional kinds of processing on double-quoted text. Since certain characters within double-quoted text are processed by the shell, they must be *escaped* within the text. Of note are the characters `$`, `` ` ``, `\`, and `"`, all of which must be preceded by a backslash within double-quoted text if they are to be passed on literally to the program. (The leading backslash is

stripped first.) Thus, the example seen previously in the section "Running awk Without Input Files" is applicable:

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

Note that the single quote is not special within double quotes.

- Null strings are removed when they occur as part of a non-null command-line argument, while explicit nonnull objects are kept. For example, to specify that the field separator `FS` should be set to the null string, use:

  ```
  awk -F "" 'program' files # correct
  ```

  Don't use this:

  ```
  awk -F"" 'program' files  # wrong!
  ```

  In the second case, *awk* will attempt to use the text of the program as the value of `FS`, and the first filename as the text of the program! This results in syntax errors at best, and confusing behavior at worst.

Mixing single and double quotes is difficult. You have to resort to shell quoting tricks, like this:

```
$ awk 'BEGIN { print "Here is a single quote <'"'"'>" }'
Here is a single quote <'>
```

This program consists of three concatenated quoted strings. The first and the third are single-quoted, the second is double-quoted.

This can be "simplified" to:

```
$ awk 'BEGIN { print "Here is a single quote <'\''>" }'
Here is a single quote <'>
```

Judge for yourself which of these two is the more readable.

Another option is to use double quotes, escaping the embedded, *awk*-level double quotes:

```
$ awk "BEGIN { print \"Here is a single quote <'>\" }"
Here is a single quote <'>
```

This option is also painful, because double quotes, backslashes, and dollar signs are very common in *awk* programs.

If you really need both single and double quotes in your *awk* program, it is probably best to move it into a separate file, where the shell won't be part of the picture, and you can say what you mean.

Datafiles for the Examples

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

In the datafile *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 only on evening and weekend hours. 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 datafile *inventory-shipped* represents information about shipments during the year. Each record contains the month, 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 last year and the first 4 months of the current 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
```

Some Simple Examples

The following command runs a simple *awk* program that searches the input file *BBS-list* for the character string `foo` (a grouping of characters is usually called a *string*; the term *string* is 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 the pattern to search for. This type of pattern is called a *regular expression*, which is covered in more detail later (see Chapter 2, *Regular Expressions*). 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 previous example and the result would be the same: all lines matching the pattern `foo` are printed. By comparison, omitting the `print` statement but retaining the curly braces makes an empty action that does nothing (i.e., no lines are printed).

Many practical *awk* programs are just a line or two. Following 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 datafile named *data*. This is just a placeholder; if you use these programs yourself, substitute your own filenames for *data*. For future reference, note that there is often more than one way to do things in *awk*. At some point, you may want to look back at these examples and see if you can come up with different ways to do the same things shown here:

- Print the length of the longest input line:

  ```
  awk '{ if (length($0) > max) max = length($0) }
       END { print max }' data
  ```

- Print every line that is longer than 80 characters:

  ```
  awk 'length($0) > 80' data
  ```

  The sole rule has a relational expression as its pattern and it has no action—so the default action, printing the record, is used.

- Print the length of the longest line in *data*:

  ```
  expand data | awk '{ if (x < length()) x = length() }
                     END { print "maximum line length is " x }'
  ```

  The input is processed by the *expand* utility to change tabs into spaces, so the widths compared are actually the right-margin columns.

- Print every line that has at least one field:

  ```
  awk 'NF > 0' data
  ```

  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 removed).

- Print seven random numbers from 0 to 100, inclusive:

  ```
  awk 'BEGIN { for (i = 1; i <= 7; i++)
                   print int(101 * rand()) }'
  ```

- Print the total number of bytes used by *files*:

  ```
  ls -l files | awk '{ x += $5 } ; END { print "total bytes: " x }'
  ```

- Print the total number of kilobytes used by *files*:

  ```
  ls -l files | awk '{ x += $5 }
                    END { print "total K-bytes: " (x + 1023)/1024 }'
  ```

- Print a sorted list of the login names of all users:

  ```
  awk -F: '{ print $1 }' /etc/passwd | sort
  ```

- Count the lines in a file:

  ```
  awk 'END { print NR }' data
  ```

- Print the even-numbered lines in the datafile:

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

  If you use the expression `NR % 2 == 1` instead, the program would print the odd-numbered lines.

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 that match the line (and perhaps there are none), *awk* reads the next line. (However, see the section "The next Statement" and also see the section "Using gawk's nextfile Statement" in Chapter 6). This continues until the program reaches the end of the file. For example, the following *awk* program contains two rules:

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

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 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 datafiles, *BBS-list* and *inventory-shipped*:

```
$ 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 beginning with `sabafoo` in *BBS-list* was printed twice, once for each rule.

A More Complex Example

Now that we've mastered some simple tasks, let's look at 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 -l | 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).* The `ls -l` part of this example is a system command that gives you a listing of the files in a directory, including each file's 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 awk.h
-rw-r--r--  1 arnold   user    983 Apr 13 12:14 awk.tab.h
-rw-r--r--  1 arnold   user  31869 Jun 15 12:20 awk.y
-rw-r--r--  1 arnold   user  22414 Nov  7 13:03 awk1.c
-rw-r--r--  1 arnold   user  37455 Nov  7 13:03 awk2.c
-rw-r--r--  1 arnold   user  27511 Dec  9 13:07 awk3.c
-rw-r--r--  1 arnold   user   7989 Nov  7 13:03 awk4.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 -l` 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's size) to the variable `sum`. As a result, when *awk* has finished reading all the input lines, `sum` is the total of the sizes of the 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 executes and prints the value of `sum`. In this example, the value of `sum` is 140963.

* In the C shell (*csh*), you need to type a semicolon and then a backslash at the end of the first line; see the section "awk Statements Versus Lines" later in this chapter for an explanation. In a POSIX-compliant shell, such as the Bourne shell or *bash*, you can type the example as shown. If the command `echo $path` produces an empty output line, you are most likely using a POSIX-compliant shell. Otherwise, you are probably using the C shell or a shell derived from it.

† On some very old systems, you may need to use `ls -lg` to get this output.

These more advanced *awk* techniques are covered in later sections (see the section "Actions" in Chapter 6). 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.

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* ignores newlines after any of the following symbols and keywords:

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

A newline at any other point is considered the end of the statement.*

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 in order to be recognized as a continuation character. A backslash is allowed 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. In *gawk*, there is no limit on the length of a line, so backslash continuation 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 entered from 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 maximum portability of your *awk* programs, it is best not to split your lines in the middle of a regular expression or a string.

* The `?` and `:` referred to here is the three-operand conditional expression described in the section "Conditional Expressions" in Chapter 5, *Expressions*. Splitting lines after `?` and `:` is a minor *gawk* extension; if *--posix* is specified (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*), then this extension is disabled.

Backslash continuation does not work as described with the C shell. It works for *awk* programs in files and for one-shot programs, provided you are using a POSIX-compliant shell, such as the Unix Bourne shell or *bash*. But the C shell 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 `>`.

Compare the previous example to how it is done with a POSIX-compliant shell:

```
$ awk 'BEGIN {
>   print \
>       "hello, world"
> }'
hello, world
```

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 option.

Another thing to keep in mind is 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
> }'
gawk: cmd. line:2:               BEGIN rule
gawk: cmd. line:2:               ^ parse error
```

In this case, it looks like the backslash would continue the comment onto the next line. However, the backslash-newline combination is never even noticed because 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. This is accomplished by separating the statements with a semicolon (`;`). This also applies to the rules themselves. Thus, the program shown at the start of this section could also be written this way:

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

The requirement that states 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.

Other Features of awk

The *awk* language provides a number of predefined, or *built-in*, variables that your programs can use to get information from *awk*. There are other variables your program can set as well 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. *gawk* provides built-in functions for working with timestamps, performing bit manipulation, and for runtime string translation.

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 the section "Built-in Variables" in Chapter 6 and the section "Built-in Functions" in Chapter 8.

When to Use awk

Now that you've seen some of what *awk* can do, you might wonder how *awk* could be useful for you. By 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 the section "A More Complex Example" earlier in this chapter.)

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. Because *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 the Glossary" 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 Java, 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.