12

The AWK Pattern Processing Language

In This Chapter

AWK is a pattern-scanning and processing language that searches one or more files for records (usually lines) that match specified patterns. It processes lines by performing actions, such as writing the record to standard output or incrementing a counter, each time it finds a match. Unlike *procedural* languages, AWK is *data driven*: You describe the data you want to work with and tell AWK what to do with the data once it finds it.

You can use AWK to generate reports or filter text. It works equally well with numbers and text; when you mix the two, AWK usually comes up with the right answer. The authors of AWK (Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan) designed the language to be easy to use. To achieve this end they sacrificed execution speed in the original implementation.

AWK takes many of its constructs from the C programming language. It includes the following features:

- A flexible format
- Conditional execution
- Looping statements
- Numeric variables
- String variables
- Regular expressions
- Relational expressions
- C's **printf**
- Coprocess execution (gawk only)
- Network data exchange (gawk only)

SYNTAX

A gawk command line has the following syntax:

*gawk [**options**] [**program**] [**file-list**]*
*gawk [**options**] –f **program-file** [**file-list**]*

The gawk utility takes its input from files you specify on the command line or from standard input. An advanced command, **getline**, gives you more choices about where input comes from and how gawk reads it (page 558). Using a coprocess, gawk can interact with another program or exchange data over a network (page 560; not available under awk or mawk). Output from gawk goes to standard output.

ARGUMENTS

In the preceding syntax, ***program*** is a gawk program that you include on the command line. The ***program-file*** is the name of the file that holds a gawk program. Putting the program on the command line allows you to write short gawk programs without having to create a separate ***program-file***. To prevent the shell from interpreting the gawk commands as shell commands, enclose the ***program*** within single quotation marks. Putting a long or complex program in a file can reduce errors and retyping.

The ***file-list*** contains the pathnames of the ordinary files that gawk processes. These files are the input files. When you do not specify a ***file-list,*** gawk takes input from standard input or as specified by **getline** (page 558) or a coprocess (page 560).

AWK has many implementations

tip The AWK language was originally implemented under UNIX as the awk utility. Most Linux distributions provide gawk (the GNU implementation of awk) or mawk (a faster, stripped-down version of awk). Mac OS X provides awk. This chapter describes gawk. All the examples in this chapter work under awk and mawk except as noted; the exceptions make use of coprocesses (page 560). You can easily install gawk on most Linux distributions. See gawk.darwinports.com if you are running Mac OS X. For a complete list of gawk extensions, see **GNU EXTENSIONS** in the gawk man page or see the gawk info page.

Options

Options preceded by a double hyphen (**––**) work under gawk only. They are not available under awk and mawk.

––field-separator ***fs***
–F ***fs***
Uses ***fs*** as the value of the input field separator (**FS** variable; page 536).

––file ***program-file***
–f ***program-file***
Reads the gawk program from the file named ***program-file*** instead of the command line. You can specify this option more than once on a command line. See page 545 for examples.

––help **–W help**
Summarizes how to use gawk (gawk only).

––lint **–W lint**
Warns about gawk constructs that may not be correct or portable (gawk only).

––posix **–W posix**
Runs a POSIX-compliant version of gawk. This option introduces some restrictions; see the gawk man page for details (gawk only).

––traditional **–W traditional**
Ignores the new GNU features in a gawk program, making the program conform to UNIX awk (gawk only).

––assign ***var=value***
–v ***var=value***
Assigns ***value*** to the variable ***var***. The assignment takes place prior to execution of the gawk program and is available within the **BEGIN** pattern (page 535). You can specify this option more than once on a command line.

Notes

See the tip on the previous page for information on AWK implementations.

For convenience many Linux systems provide a link from **/bin/awk** to **/bin/gawk** or **/bin/mawk**. As a result you can run the program using either name.

Language Basics

A gawk program (from ***program*** on the command line or from ***program-file***) consists of one or more lines containing a ***pattern*** and/or ***action*** in the following format:

pattern *{* ***action*** *}*

The ***pattern*** selects lines from the input. The gawk utility performs the ***action*** on all lines that the ***pattern*** selects. The braces surrounding the ***action*** enable gawk to differentiate it from the ***pattern***. If a program line does not contain a ***pattern***, gawk selects all lines in the input. If a program line does not contain an ***action***, gawk copies the selected lines to standard output.

To start, gawk compares the first line of input (from the ***file-list*** or standard input) with each ***pattern*** in the program. If a ***pattern*** selects the line (if there is a match), gawk takes the ***action*** associated with the ***pattern***. If the line is not selected, gawk does not take the ***action***. When gawk has completed its comparisons for the first line of input, it repeats the process for the next line of input. It continues this process of comparing subsequent lines of input until it has read all of the input.

If several ***patterns*** select the same line, gawk takes the ***actions*** associated with each of the ***patterns*** in the order in which they appear in the program. It is possible for gawk to send a single line from the input to standard output more than once.

Patterns

~ and !~ You can use a regular expression (Appendix A), enclosed within slashes, as a ***pattern***. The ~ operator tests whether a field or variable matches a regular expression (examples on page 543). The **!~** operator tests for no match. You can perform both numeric and string comparisons using the relational operators listed in Table 12-1. You can combine any of the ***patterns*** using the Boolean operators || (OR) or && (AND).

Table 12-1 Relational operators

Relational operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to

Table 12-1 Relational operators (continued)

Relational operator	Meaning
!=	Not equal to
>=	Greater than or equal to
>	Greater than

BEGIN and END Two unique ***patterns***, **BEGIN** and **END**, execute commands before gawk starts processing the input and after it finishes processing the input. The gawk utility executes the ***actions*** associated with the **BEGIN** ***pattern*** before, and with the **END** ***pattern*** after, it processes all the input. See pages 545 and 546 for examples.

, (comma) The comma is the range operator. If you separate two ***patterns*** with a comma on a single gawk program line, gawk selects a range of lines, beginning with the first line that matches the first ***pattern***. The last line selected by gawk is the next subsequent line that matches the second ***pattern***. If no line matches the second ***pattern***, gawk selects every line through the end of the input. After gawk finds the second ***pattern***, it begins the process again by looking for the first ***pattern*** again. See page 544 for examples.

Actions

The ***action*** portion of a gawk command causes gawk to take that ***action*** when it matches a ***pattern***. When you do not specify an ***action***, gawk performs the default ***action***, which is the **print** command (explicitly represented as {**print**}). This ***action*** copies the record (normally a line; see "Record separators" on the next page) from the input to standard output.

When you follow a **print** command with arguments, gawk displays only the arguments you specify. These arguments can be variables or string constants. You can send the output from a **print** command to a file (use > within the gawk program; page 549), append it to a file (>>), or send it through a pipe to the input of another program (|). A coprocess (|&) is a two-way pipe that exchanges data with a program running in the background (available under gawk only; page 560).

Unless you separate items in a **print** command with commas, gawk catenates them. Commas cause gawk to separate the items with the output field separator (**OFS**, normally a SPACE; page 536).

You can include several ***actions*** on one line by separating them with semicolons.

Comments

The gawk utility disregards anything on a program line following a pound sign (#). You can document a gawk program by preceding comments with this symbol.

Variables

Although you do not need to declare gawk variables prior to their use, you can assign initial values to them if you like. Unassigned numeric variables are initialized to 0; string variables are initialized to the null string. In addition to supporting user variables, gawk maintains program variables. You can use both user and program variables in the ***pattern*** and ***action*** portions of a gawk program. Table 12-2 lists a few program variables.

Table 12-2 Variables

Variable	Meaning
$0	The current record (as a single variable)
$1–$*n*	Fields in the current record
FILENAME	Name of the current input file (null for standard input)
FS	Input field separator (default: SPACE or TAB; page 550)
NF	Number of fields in the current record (page 554)
NR	Record number of the current record (page 546)
OFS	Output field separator (default: SPACE; page 547)
ORS	Output record separator (default: NEWLINE; page 554)
RS	Input record separator (default: NEWLINE)

In addition to initializing variables within a program, you can use the **––assign** (**–v**) option to initialize variables on the command line. This feature is useful when the value of a variable changes from one run of gawk to the next.

Record separators

By default the input and output record separators are NEWLINE characters. Thus gawk takes each line of input to be a separate record and appends a NEWLINE to the end of each output record. By default the input field separators are SPACEs and TABs; the default output field separator is a SPACE. You can change the value of any of the separators at any time by assigning a new value to its associated variable either from within the program or from the command line by using the **––assign** (**–v**) option.

Functions

Table 12-3 lists a few of the functions gawk provides for manipulating numbers and strings.

Table 12-3 Functions

Function	Meaning
length(*str*)	Returns the number of characters in ***str;*** without an argument, returns the number of characters in the current record (page 545)
int(*num*)	Returns the integer portion of ***num***
index(*str1*,*str2*)	Returns the index of ***str2*** in ***str1*** or 0 if ***str2*** is not present
split(*str*,*arr*,*del*)	Places elements of ***str***, delimited by ***del***, in the array ***arr***[1]...***arr***[***n***]; returns the number of elements in the array (page 556)

Table 12-3 Functions (continued)

Function	Meaning
sprintf(*fmt*,*args*)	Formats ***args*** according to ***fmt*** and returns the formatted string; mimics the C programming language function of the same name
substr(*str*,*pos*,*len*)	Returns the substring of ***str*** that begins at ***pos*** and is ***len*** characters long
tolower(*str*)	Returns a copy of ***str*** in which all uppercase letters are replaced with their lowercase counterparts
toupper(*str*)	Returns a copy of ***str*** in which all lowercase letters are replaced with their uppercase counterparts

Arithmetic Operators

The gawk arithmetic operators listed in Table 12-4 are from the C programming language.

Table 12-4 Arithmetic operators

Operator	Meaning
**	Raises the expression preceding the operator to the power of the expression following it
*	Multiplies the expression preceding the operator by the expression following it
/	Divides the expression preceding the operator by the expression following it
%	Takes the remainder after dividing the expression preceding the operator by the expression following it
+	Adds the expression preceding the operator to the expression following it
–	Subtracts the expression following the operator from the expression preceding it
=	Assigns the value of the expression following the operator to the variable preceding it
++	Increments the variable preceding the operator
––	Decrements the variable preceding the operator
+=	Adds the expression following the operator to the variable preceding it and assigns the result to the variable preceding the operator
–=	Subtracts the expression following the operator from the variable preceding it and assigns the result to the variable preceding the operator
*=	Multiplies the variable preceding the operator by the expression following it and assigns the result to the variable preceding the operator
/=	Divides the variable preceding the operator by the expression following it and assigns the result to the variable preceding the operator
%=	Assigns the remainder, after dividing the variable preceding the operator by the expression following it, to the variable preceding the operator

Associative Arrays

The *associative array* is one of gawk's most powerful features. These arrays use strings as indexes. Using an associative array, you can mimic a traditional array by using numeric strings as indexes. In Perl, an associative array is called a *hash* (page 500).

You assign a value to an element of an associative array using the following syntax:

array[string] = value

where ***array*** is the name of the array, ***string*** is the index of the element of the array you are assigning a value to, and ***value*** is the value you are assigning to that element.

Using the following syntax, you can use a **for** structure with an associative array:

*for (**elem** in **array**) **action***

where ***elem*** is a variable that takes on the value of each element of the array as the **for** structure loops through them, ***array*** is the name of the array, and ***action*** is the action that gawk takes for each element in the array. You can use the ***elem*** variable in this ***action***.

See page 551 for example programs that use associative arrays.

printf

You can use the **printf** command in place of **print** to control the format of the output gawk generates. The gawk version of **printf** is similar to that found in the C language. A **printf** command has the following syntax:

*printf "**control-string**", **arg1, arg2, ..., argn***

The ***control-string*** determines how **printf** formats ***arg1, arg2, ..., argn***. These arguments can be variables or other expressions. Within the ***control-string*** you can use **\n** to indicate a NEWLINE and **\t** to indicate a TAB. The ***control-string*** contains conversion specifications, one for each argument. A conversion specification has the following syntax:

*%[–][**x**[.**y**]]**conv***

where **–** causes **printf** to left-justify the argument, ***x*** is the minimum field width, and ***.y*** is the number of places to the right of a decimal point in a number. The ***conv*** indicates the type of numeric conversion and can be selected from the letters in Table 12-5. See page 548 for example programs that use **printf**.

Table 12-5 Numeric conversion

conv	**Type of conversion**
d	Decimal
e	Exponential notation
f	Floating-point number

Table 12-5 Numeric conversion (continued)

conv	**Type of conversion**
g	Use **f** or **e**, whichever is shorter
o	Unsigned octal
s	String of characters
x	Unsigned hexadecimal

Control Structures

Control (flow) statements alter the order of execution of commands within a gawk program. This section details the **if...else**, **while**, and **for** control structures. In addition, the **break** and **continue** statements work in conjunction with the control structures to alter the order of execution of commands. See page 398 for more information on control structures. You do not need to use braces around ***commands*** when you specify a single, simple command.

if...else

The **if...else** control structure tests the status returned by the ***condition*** and transfers control based on this status. The syntax of an **if...else** structure is shown below. The **else** part is optional.

```
if (condition)
        {commands}
    [else
        {commands}]
```

The simple **if** statement shown here does not use braces:

```
if ($5 <= 5000) print $0
```

Next is a gawk program that uses a simple **if...else** structure. Again, there are no braces.

```
$ cat if1
BEGIN   {
        nam="sam"
        if (nam == "max")
                print "nam is max"
            else
                print "nam is not max, it is", nam
        }
$ gawk -f if1
nam is not max, it is sam
```

while

The **while** structure loops through and executes the ***commands*** as long as the ***condition*** is *true*. The syntax of a **while** structure is

```
while (condition)
    {commands}
```

The next gawk program uses a simple **while** structure to display powers of 2. This example uses braces because the **while** loop contains more than one statement. This program does not accept input; all processing takes place when gawk executes the statements associated with the BEGIN pattern.

```
$ cat while1
BEGIN{
    n = 1
    while (n <= 5)
        {
        print "2^" n, 2**n
        n++
        }
    }

$ gawk -f while1
1^2 2
2^2 4
3^2 8
4^2 16
5^2 32
```

for

The syntax of a **for** control structure is

> *for (**init; condition; increment**)*
>   *{**commands**}*

A **for** structure starts by executing the ***init*** statement, which usually sets a counter to 0 or 1. It then loops through the ***commands*** as long as the ***condition*** remains *true*. After each loop it executes the ***increment*** statement. The **for1** gawk program does the same thing as the preceding **while1** program except that it uses a **for** statement, which makes the program simpler:

```
$ cat for1
BEGIN   {
        for (n=1; n <= 5; n++)
        print "2^" n, 2**n
        }

$ gawk -f for1
1^2 2
2^2 4
3^2 8
4^2 16
5^2 32
```

The gawk utility supports an alternative **for** syntax for working with associative arrays:

> *for (**var in array**)*
>   *{**commands**}*

This **for** structure loops through elements of the associative array named ***array***, assigning the value of the index of each element of ***array*** to ***var*** each time through the loop. The following line of code (from the program on page 551) demonstrates a **for** structure:

```
END    {for (name in manuf) print name, manuf[name]}
```

break

The **break** statement transfers control out of a **for** or **while** loop, terminating execution of the innermost loop it appears in.

continue

The **continue** statement transfers control to the end of a **for** or **while** loop, causing execution of the innermost loop it appears in to continue with the next iteration.

Examples

cars data file Many of the examples in this section work with the **cars** data file. From left to right, the columns in the file contain each car's make, model, year of manufacture, mileage in thousands of miles, and price. All whitespace in this file is composed of single TABs (the file does not contain any SPACEs).

```
$ cat cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
toyota  rav4    2002    180     750
chevy   impala  1985    85      1550
ford    explor  2003    25      9500
```

Missing pattern A simple gawk program is

```
{ print }
```

This program consists of one program line that is an ***action***. Because the ***pattern*** is missing, gawk selects all lines of input. When used without any arguments the **print** command displays each selected line in its entirety. This program copies the input to standard output.

```
$ gawk '{ print }' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

Missing action The next program has a ***pattern*** but no explicit ***action.*** The slashes indicate that **chevy** is a regular expression.

```
/chevy/
```

In this case gawk selects from the input just those lines that contain the string **chevy**. When you do not specify an ***action,*** gawk assumes the ***action*** is **print**. The following example copies to standard output all lines from the input that contain the string **chevy:**

```
$ gawk '/chevy/' cars
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
chevy   impala  1985    85      1550
```

Single quotation marks Although neither gawk nor shell syntax requires single quotation marks on the command line, it is still a good idea to use them because they can prevent problems. If the gawk program you create on the command line includes SPACEs or characters that are special to the shell, you must quote them. Always enclosing the program in single quotation marks is the easiest way to make sure you have quoted any characters that need to be quoted.

Fields The next example selects all lines from the file (it has no ***pattern***). The braces enclose the ***action;*** you must always use braces to delimit the ***action*** so gawk can distinguish it from the ***pattern.*** This example displays the third field (**$3**), a SPACE (the output field separator, indicated by the comma), and the first field (**$1**) of each selected line:

```
$ gawk '{print $3, $1}' cars
1970 plym
1999 chevy
1965 ford
1998 volvo
...
```

The next example, which includes both a ***pattern*** and an ***action,*** selects all lines that contain the string **chevy** and displays the third and first fields from the selected lines:

```
$ gawk '/chevy/ {print $3, $1}' cars
1999 chevy
2000 chevy
1985 chevy
```

In the following example, gawk selects lines that contain a match for the regular expression **h**. Because there is no explicit ***action,*** gawk displays all the lines it selects.

```
$ gawk '/h/' cars
chevy   malibu  1999    60      3000
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
chevy   impala  1985    85      1550
```

~ (matches operator) The next ***pattern*** uses the matches operator (~) to select all lines that contain the letter **h** in the first field:

```
$ gawk '$1 ~ /h/' cars
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
chevy   impala  1985    85      1550
```

The caret (^) in a regular expression forces a match at the beginning of the line (page 890) or, in this case, at the beginning of the first field:

```
$ gawk '$1 ~ /^h/' cars
honda   accord  2001    30      6000
```

Brackets surround a character class definition (page 889). In the next example, gawk selects lines that have a second field that begins with **t** or **m** and displays the third and second fields, a dollar sign, and the fifth field. Because there is no comma between the "$" and the $5, gawk does not put a SPACE between them in the output.

```
$ gawk '$2 ~ /^[tm]/ {print $3, $2, "$"  $5}' cars
1999 malibu $3000
1965 mustang $10000
2003 thundbd $10500
2000 malibu $3500
2004 taurus $17000
```

Dollar signs The next example shows three roles a dollar sign can play in a gawk program. First, a dollar sign followed by a number names a field. Second, within a regular expression a dollar sign forces a match at the end of a line or field (5$). Third, within a string a dollar sign represents itself.

```
$ gawk '$3 ~ /5$/ {print $3, $1, "$"  $5}' cars
1965 ford $10000
1985 bmw $450
1985 chevy $1550
```

In the next example, the equal-to relational operator (==) causes gawk to perform a numeric comparison between the third field in each line and the number **1985**. The gawk command takes the default ***action***, **print**, on each line where the comparison is *true*.

```
$ gawk '$3 == 1985' cars
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
```

The next example finds all cars priced at or less than $3,000.

```
$ gawk '$5 <= 3000' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
bmw     325i    1985    115     450
toyota  rav4    2002    180     750
chevy   impala  1985    85      1550
```

Textual comparisons

When you use double quotation marks, gawk performs textual comparisons by using the ASCII (or other local) collating sequence as the basis of the comparison. In the following example, gawk shows that the *strings* **450** and **750** fall in the range that lies between the *strings* **2000** and **9000**, which is probably not the intended result.

```
$ gawk '"2000" <= $5 && $5 < "9000"' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
toyota  rav4    2002    180     750
```

When you need to perform a numeric comparison, do not use quotation marks. The next example gives the intended result. It is the same as the previous example except it omits the double quotation marks.

```
$ gawk '2000 <= $5 && $5 < 9000' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
```

, (range operator)

The range operator (**,**) selects a group of lines. The first line it selects is the one specified by the ***pattern*** before the comma. The last line is the one selected by the ***pattern*** after the comma. If no line matches the ***pattern*** after the comma, gawk selects every line through the end of the input. The next example selects all lines, starting with the line that contains **volvo** and ending with the line that contains **bmw**.

```
$ gawk '/volvo/ , /bmw/' cars
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
```

After the range operator finds its first group of lines, it begins the process again, looking for a line that matches the ***pattern*** before the comma. In the following example, gawk finds three groups of lines that fall between **chevy** and **ford**. Although the fifth line of input contains **ford**, gawk does not select it because at the time it is processing the fifth line, it is searching for **chevy**.

```
$ gawk '/chevy/ , /ford/' cars
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
```

```
ford    taurus  2004    10      17000
chevy   impala  1985    85      1550
ford    explor  2003    25      9500
```

––file option When you are writing a longer gawk program, it is convenient to put the program in a file and reference the file on the command line. Use the **–f** (**––file**) option followed by the name of the file containing the gawk program.

BEGIN The following gawk program, which is stored in a file named **pr_header**, has two ***actions*** and uses the **BEGIN** ***pattern***. The gawk utility performs the ***action*** associated with **BEGIN** before processing any lines of the data file: It displays a header. The second ***action***, {**print**}, has no ***pattern*** part and displays all lines from the input.

```
$ cat pr_header
BEGIN   {print "Make    Model   Year    Miles   Price"}
        {print}

$ gawk -f pr_header cars
Make    Model   Year    Miles   Price
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

The next example expands the ***action*** associated with the **BEGIN** ***pattern***. In the previous and the following examples, the whitespace in the headers is composed of single TABs, so the titles line up with the columns of data.

```
$ cat pr_header2
BEGIN   {
print "Make     Model   Year    Miles   Price"
print "----------------------------------------"
}
        {print}

$ gawk -f pr_header2 cars
Make    Model   Year    Miles   Price
----------------------------------------
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

length function When you call the **length** function without an argument, it returns the number of characters in the current line, including field separators. The **$0** variable always contains the value of the current line. In the next example, gawk prepends the line length to each line and then a pipe sends the output from gawk to sort (the **–n** option specifies a numeric sort; page 817). As a result, the lines of the **cars** file appear in order of line length.

```
$ gawk '{print length, $0}' cars | sort -n
21 bmw  325i    1985    115     450
22 plym fury    1970    73      2500
23 volvo        s80     1998    102     9850
24 ford explor  2003    25      9500
24 toyota       rav4    2002    180     750
25 chevy        impala  1985    85      1550
25 chevy        malibu  1999    60      3000
25 chevy        malibu  2000    50      3500
25 ford taurus  2004    10      17000
25 honda        accord  2001    30      6000
26 ford mustang 1965    45      10000
26 ford thundbd 2003    15      10500
```

The formatting of this report depends on TABs for horizontal alignment. The three extra characters at the beginning of each line throw off the format of several lines; a remedy for this situation is covered shortly.

NR (record number)

The **NR** variable contains the record (line) number of the current line. The following ***pattern*** selects all lines that contain more than 24 characters. The ***action*** displays the line number of each of the selected lines.

```
$ gawk 'length > 24 {print NR}' cars
2
3
5
6
8
9
11
```

You can combine the range operator (**,**) and the **NR** variable to display a group of lines of a file based on their line numbers. The next example displays lines 2 through 4:

```
$ gawk 'NR == 2 , NR == 4' cars
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
```

END

The **END** ***pattern*** works in a manner similar to the **BEGIN** ***pattern***, except gawk takes the ***actions*** associated with this pattern after processing the last line of input. The following report displays information only after it has processed all the input. The **NR** variable retains its value after gawk finishes processing the data file, so an ***action*** associated with an **END** ***pattern*** can use it.

```
$ gawk 'END {print NR, "cars for sale." }' cars
12 cars for sale.
```

The next example uses **if** control structures to expand the abbreviations used in some of the first fields. As long as gawk does not change a record, it leaves the entire record—including any separators—intact. Once it makes a change to a record, gawk changes all separators in that record to the value of the output field separator. The default output field separator is a SPACE.

```
$ cat separ_demo
        {
        if ($1 ~ /ply/)  $1 = "plymouth"
        if ($1 ~ /chev/) $1 = "chevrolet"
        print
        }

$ gawk -f separ_demo cars
plymouth fury 1970 73 2500
chevrolet malibu 1999 60 3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevrolet malibu 2000 50 3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
toyota  rav4    2002    180     750
chevrolet impala 1985 85 1550
ford    explor  2003    25      9500
```

Stand-alone script Instead of calling gawk from the command line with the **–f** option and the name of the program you want to run, you can write a script that calls gawk with the commands you want to run. The next example is a stand-alone script that runs the same program as the previous example. The **#!/bin/gawk –f** command (page 280) runs the gawk utility directly. To execute it, you need both read and execute permission to the file holding the script (page 278).

```
$ chmod u+rx separ_demo2
$ cat separ_demo2
#!/bin/gawk -f
        {
        if ($1 ~ /ply/)  $1 = "plymouth"
        if ($1 ~ /chev/) $1 = "chevrolet"
        print
        }

$ ./separ_demo2 cars
plymouth fury 1970 73 2500
chevrolet malibu 1999 60 3000
ford    mustang 1965    45      10000
...
```

OFS variable You can change the value of the output field separator by assigning a value to the **OFS** variable. The following example assigns a TAB character to **OFS**, using the backslash escape sequence **\t**. This fix improves the appearance of the report but does not line up the columns properly.

```
$ cat ofs_demo
BEGIN   {OFS = "\t"}
        {
        if ($1 ~ /ply/)  $1 = "plymouth"
        if ($1 ~ /chev/) $1 = "chevrolet"
        print
        }

$ gawk -f ofs_demo cars
plymouth        fury    1970    73      2500
chevrolet       malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevrolet       malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
toyota  rav4    2002    180     750
chevrolet       impala  1985    85      1550
ford    explor  2003    25      9500
```

printf You can use **printf** (page 538) to refine the output format. The following example uses a backslash at the end of two program lines to quote the following NEWLINE. You can use this technique to continue a long line over one or more lines without affecting the outcome of the program.

```
$ cat printf_demo
BEGIN   {
    print "                                 Miles"
    print "Make       Model       Year      (000)       Price"
    print \
    "--------------------------------------------------"
    }
    {
    if ($1 ~ /ply/)  $1 = "plymouth"
    if ($1 ~ /chev/) $1 = "chevrolet"
    printf "%-10s %-8s    %2d   %5d     $ %8.2f\n",\
        $1, $2, $3, $4, $5
    }

$ gawk -f printf_demo cars
                                 Miles
Make       Model       Year      (000)       Price
--------------------------------------------------
plymouth   fury        1970         73     $  2500.00
chevrolet  malibu      1999         60     $  3000.00
ford       mustang     1965         45     $ 10000.00
volvo      s80         1998        102     $  9850.00
ford       thundbd     2003         15     $ 10500.00
chevrolet  malibu      2000         50     $  3500.00
bmw        325i        1985        115     $   450.00
```

```
honda      accord     2001     30     $  6000.00
ford       taurus     2004     10     $ 17000.00
toyota     rav4       2002    180     $   750.00
chevrolet  impala     1985     85     $  1550.00
ford       explor     2003     25     $  9500.00
```

Redirecting output The next example creates two files: one with the lines that contain **chevy** and one with the lines that contain **ford**.

```
$ cat redirect_out
/chevy/    {print > "chevfile"}
/ford/     {print > "fordfile"}
END        {print "done."}

$ gawk -f redirect_out cars
done.

$ cat chevfile
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
chevy   impala  1985    85      1550
```

The **summary** program produces a summary report on all cars and newer cars. Although they are not required, the initializations at the beginning of the program represent good programming practice; gawk automatically declares and initializes variables as you use them. After reading all the input data, gawk computes and displays the averages.

```
$ cat summary
BEGIN   {
        yearsum = 0 ; costsum = 0
        newcostsum = 0 ; newcount = 0
        }
        {
        yearsum += $3
        costsum += $5
        }
$3 > 2000 {newcostsum += $5 ; newcount ++}
END     {
        printf "Average age of cars is %4.1f years\n",\
            2006 - (yearsum/NR)
        printf "Average cost of cars is $%7.2f\n",\
            costsum/NR
            printf "Average cost of newer cars is $%7.2f\n",\
                newcostsum/newcount
        }

$ gawk -f summary cars
Average age of cars is 13.1 years
Average cost of cars is $6216.67
Average cost of newer cars is $8750.00
```

The following gawk command shows the format of a line from a Linux **passwd** file that the next example uses:

```
$ awk '/mark/ {print}' /etc/passwd
mark:x:107:100:ext 112:/home/mark:/bin/tcsh
```

FS variable

The next example demonstrates a technique for finding the largest number in a field. Because it works with a Linux **passwd** file, which delimits fields with colons (**:**), the example changes the input field separator (**FS**) before reading any data. It reads the **passwd** file and determines the next available user ID number (field 3). The numbers do not have to be in order in the **passwd** file for this program to work.

The ***pattern*** (**$3 > saveit**) causes gawk to select records that contain a user ID number greater than any previous user ID number it has processed. Each time it selects a record, gawk assigns the value of the new user ID number to the **saveit** variable. Then gawk uses the new value of **saveit** to test the user IDs of all subsequent records. Finally gawk adds 1 to the value of **saveit** and displays the result.

```
$ cat find_uid
BEGIN          {FS = ":"
               saveit = 0}
$3 > saveit    {saveit = $3}
END            {print "Next available UID is " saveit + 1}

$ gawk -f find_uid /etc/passwd
Next available UID is 1092
```

The next example produces another report based on the **cars** file. This report uses nested **if...else** control structures to substitute values based on the contents of the price field. The program has no ***pattern*** part; it processes every record.

```
$ cat price_range
    {
    if             ($5 <= 5000)               $5 = "inexpensive"
        else if    (5000 < $5 && $5 < 10000)  $5 = "please ask"
        else if    (10000 <= $5)              $5 = "expensive"
    #
    printf "%-10s %-8s    %2d    %5d    %-12s\n",\
    $1, $2, $3, $4, $5
    }

$ gawk -f price_range cars
plym       fury        1970       73    inexpensive
chevy      malibu      1999       60    inexpensive
ford       mustang     1965       45    expensive
volvo      s80         1998      102    please ask
ford       thundbd     2003       15    expensive
chevy      malibu      2000       50    inexpensive
bmw        325i        1985      115    inexpensive
honda      accord      2001       30    please ask
ford       taurus      2004       10    expensive
toyota     rav4        2002      180    inexpensive
chevy      impala      1985       85    inexpensive
ford       explor      2003       25    please ask
```

Associative arrays Next the **manuf** associative array uses the contents of the first field of each record in the **cars** file as an index. The array consists of the elements **manuf[plym]**, **manuf[chevy]**, **manuf[ford]**, and so on. Each new element is initialized to 0 (zero) as it is created. The **++** operator increments the variable it follows.

for structure The ***action*** following the **END** ***pattern*** is a **for** structure, which loops through the elements of an associative array. A pipe sends the output through sort to produce an alphabetical list of cars and the quantities in stock. Because it is a shell script and not a gawk program file, you must have both read and execute permission to the **manuf** file to execute it as a command.

```
$ cat manuf
gawk '  {manuf[$1]++}
END     {for (name in manuf) print name, manuf[name]}
' cars |
sort

$ ./manuf
bmw 1
chevy 3
ford 4
honda 1
plym 1
toyota 1
volvo 1
```

The next program, **manuf.sh**, is a more general shell script that includes error checking. This script lists and counts the contents of a column in a file, with both the column number and the name of the file specified on the command line.

The first ***action*** (the one that starts with {**count**) uses the shell variable **$1** in the middle of the gawk program to specify an array index. Because of the way the single quotation marks are paired, the **$1** that appears to be within single quotation marks is actually not quoted: The two quoted strings in the gawk program surround, but do not include, the **$1**. Because the **$1** is not quoted, and because this is a shell script, the shell substitutes the value of the first command-line argument in place of **$1** (page 441). As a result, the **$1** is interpreted before the gawk command is invoked. The leading dollar sign (the one before the first single quotation mark on that line) causes gawk to interpret what the shell substitutes as a field number.

```
$ cat manuf.sh
if [ $# != 2 ]
    then
        echo "Usage: manuf.sh field file"
        exit 1
fi
gawk < $2 '
        {count[$'$1']++}
END     {for (item in count) printf "%-20s%-20s\n",\
            item, count[item]}' |
sort
```

```
$ ./manuf.sh
Usage: manuf.sh field file

$ ./manuf.sh 1 cars
bmw                 1
chevy               3
ford                4
honda               1
plym                1
toyota              1
volvo               1

$ ./manuf.sh 3 cars
1965                1
1970                1
1985                2
1998                1
1999                1
2000                1
2001                1
2002                1
2003                2
2004                1
```

A way around the tricky use of quotation marks that allow parameter expansion within the gawk program is to use the **–v** option on the command line to pass the field number to gawk as a variable. This change makes it easier for someone else to read and debug the script. You call the **manuf2.sh** script the same way you call **manuf.sh**:

```
$ cat manuf2.sh
if [ $# != 2 ]
        then
                echo "Usage: manuf.sh field file"
                exit 1
fi
gawk -v "field=$1" < $2 '
                {count[$field]++}
END             {for (item in count) printf "%-20s%-20s\n",\
                        item, count[item]}' |
sort
```

The **word_usage** script displays a word usage list for a file you specify on the command line. The tr utility (page 864) lists the words from standard input, one to a line. The sort utility orders the file, putting the most frequently used words first. The script sorts groups of words that are used the same number of times in alphabetical order.

```
$ cat word_usage
tr -cs 'a-zA-Z' '[\n*]' < $1 |
gawk    '
        {count[$1]++}
END     {for (item in count) printf "%-15s%3s\n", item, count[item]}' |
sort +1nr +0f -1
```

```
$ ./word_usage textfile
the             42
file            29
fsck            27
system          22
you             22
to              21
it              17
SIZE            14
and             13
MODE            13
...
```

Following is a similar program in a different format. The style mimics that of a C program and may be easier to read and work with for more complex gawk programs.

```
$ cat word_count
tr -cs 'a-zA-Z' '[\n*]' < $1 |
gawk '  {
        count[$1]++
}
END     {
        for (item in count)
            {
            if (count[item] > 4)
                {
                printf "%-15s%3s\n", item, count[item]
                }
        }
} ' |
sort +1nr +0f -1
```

The tail utility displays the last ten lines of output, illustrating that words occurring fewer than five times are not listed:

```
$ ./word_count textfile | tail
directories      5
if               5
information      5
INODE            5
more             5
no               5
on               5
response         5
this             5
will             5
```

The next example shows one way to put a date on a report. The first line of input to the gawk program comes from date. The program reads this line as record number 1 (**NR == 1**), processes it accordingly, and processes all subsequent lines with the ***action*** associated with the next ***pattern*** (**NR > 1**).

```
$ cat report
if (test $# = 0) then
    echo "You must supply a filename."
    exit 1
fi
(date; cat $1) |
gawk '
NR == 1    {print "Report for", $1, $2, $3 ", " $6}
NR >  1    {print $5 "\t" $1}'

$ ./report cars
Report for Mon Jan 31, 2010
2500    plym
3000    chevy
10000   ford
9850    volvo
10500   ford
3500    chevy
450     bmw
6000    honda
17000   ford
750     toyota
1550    chevy
9500    ford
```

The next example sums each of the columns in a file you specify on the command line; it takes its input from the **numbers** file. The program performs error checking, reporting on and discarding rows that contain nonnumeric entries. It uses the **next** command (with the comment **skip bad records**) to skip the rest of the commands for the current record if the record contains a nonnumeric entry. At the end of the program, gawk displays a grand total for the file.

```
$ cat numbers
10      20      30.3    40.5
20      30      45.7    66.1
30      xyz     50      70
40      75      107.2   55.6
50      20      30.3    40.5
60      30      45.0    66.1
70      1134.7  50      70
80      75      107.2   55.6
90      176     30.3    40.5
100     1027.45 45.7    66.1
110     123     50      57a.5
120     75      107.2   55.6
```

```
$ cat tally
gawk ' BEGIN   {
               ORS = ""
               }

NR == 1 {                                        # first record only
    nfields = NF                                 # set nfields to number of
    }                                            # fields in the record (NF)
    {
```

```
    if ($0 ~ /[^0-9. \t]/)                       # check each record to see if it contains
        {                                        # any characters that are not numbers,
        print "\nRecord " NR " skipped:\n\t"      # periods, spaces, or TABs
        print $0 "\n"
        next                                     # skip bad records
        }
    else
        {
        for (count = 1; count <= nfields; count++) # for good records loop through fields
            {
            printf "%10.2f", $count > "tally.out"
            sum[count] += $count
            gtotal += $count
            }
        print "\n" > "tally.out"
        }
    }

END     {                                        # after processing last record
    for (count = 1; count <= nfields; count++)   # print summary
        {
        print "   -------" > "tally.out"
        }
    print "\n" > "tally.out"
    for (count = 1; count <= nfields; count++)
        {
        printf "%10.2f", sum[count] > "tally.out"
        }
    print "\n\n        Grand Total " gtotal "\n" > "tally.out"
} ' < numbers
```

```
$ ./tally
Record 3 skipped:
        30      xyz     50      70

Record 6 skipped:
        60      30      45.O    66.1

Record 11 skipped:
        110     123     50      57a.5

$ cat tally.out
     10.00     20.00     30.30     40.50
     20.00     30.00     45.70     66.10
     40.00     75.00    107.20     55.60
     50.00     20.00     30.30     40.50
     70.00   1134.70     50.00     70.00
     80.00     75.00    107.20     55.60
     90.00    176.00     30.30     40.50
    100.00   1027.45     45.70     66.10
    120.00     75.00    107.20     55.60
   -------   -------   -------   -------
    580.00   2633.15    553.90    490.50

        Grand Total 4257.55
```

The next example reads the **passwd** file, listing users who do not have passwords and users who have duplicate user ID numbers. (The pwck utility [Linux only] performs similar checks.) Because Mac OS X uses Open Directory (page 926) and not the **passwd** file, this example will not work under OS X.

```
$ cat /etc/passwd
bill::102:100:ext 123:/home/bill:/bin/bash
roy:x:104:100:ext 475:/home/roy:/bin/bash
tom:x:105:100:ext 476:/home/tom:/bin/bash
lynn:x:166:100:ext 500:/home/lynn:/bin/bash
mark:x:107:100:ext 112:/home/mark:/bin/bash
sales:x:108:100:ext 102:/m/market:/bin/bash
anne:x:109:100:ext 355:/home/anne:/bin/bash
toni::164:100:ext 357:/home/toni:/bin/bash
ginny:x:115:100:ext 109:/home/ginny:/bin/bash
chuck:x:116:100:ext 146:/home/chuck:/bin/bash
neil:x:164:100:ext 159:/home/neil:/bin/bash
rmi:x:118:100:ext 178:/home/rmi:/bin/bash
vern:x:119:100:ext 201:/home/vern:/bin/bash
bob:x:120:100:ext 227:/home/bob:/bin/bash
janet:x:122:100:ext 229:/home/janet:/bin/bash
maggie:x:124:100:ext 244:/home/maggie:/bin/bash
dan::126:100::/home/dan:/bin/bash
dave:x:108:100:ext 427:/home/dave:/bin/bash
mary:x:129:100:ext 303:/home/mary:/bin/bash
```

```
$ cat passwd_check
gawk < /etc/passwd '     BEGIN   {
    uid[void] = ""                           # tell gawk that uid is an array
    }
    {                                        # no pattern indicates process all records
    dup = 0                                  # initialize duplicate flag
    split($0, field, ":")                    # split into fields delimited by ":"
    if (field[2] == "")                      # check for null password field
        {
        if (field[5] == "")                  # check for null info field
            {
            print field[1] " has no password."
            }
        else
            {
            print field[1] " ("field[5]") has no password."
            }
        }
    for (name in uid)                        # loop through uid array
        {
        if (uid[name] == field[3])           # check for second use of UID
            {
            print field[1] " has the same UID as " name " : UID = " uid[name]
            dup = 1                          # set duplicate flag
            }
        }
if (!dup)                                # same as if (dup == 0)
                                         # assign UID and login name to uid array
    {
    uid[field[1]] = field[3]
    }
}'
```

```
$ ./passwd_check
bill (ext 123) has no password.
toni (ext 357) has no password.
neil has the same UID as toni : UID = 164
dan has no password.
dave has the same UID as sales : UID = 108
```

The next example shows a complete interactive shell script that uses gawk to generate a report on the **cars** file based on price ranges:

```
$ cat list_cars
trap 'rm -f $$.tem > /dev/null;echo $0 aborted.;exit 1' 1 2 15
echo -n "Price range (for example, 5000 7500):"
read lowrange hirange

echo '
                               Miles
Make       Model       Year    (000)         Price
--------------------------------------------------' > $$.tem
gawk < cars '
$5 >= '$lowrange' && $5 <= '$hirange' {
        if ($1 ~ /ply/)  $1 = "plymouth"
        if ($1 ~ /chev/) $1 = "chevrolet"
        printf "%-10s %-8s    %2d    %5d    $ %8.2f\n", $1, $2, $3, $4,
$5
        }' | sort -n +5 >> $$.tem
cat $$.tem
rm $$.tem

$ ./list_cars
Price range (for example, 5000 7500):3000 8000

                               Miles
Make       Model       Year    (000)         Price
--------------------------------------------------
chevrolet  malibu      1999       60    $  3000.00
chevrolet  malibu      2000       50    $  3500.00
honda      accord      2001       30    $  6000.00

$ ./list_cars
Price range (for example, 5000 7500):0 2000

                               Miles
Make       Model       Year    (000)         Price
--------------------------------------------------
bmw        325i        1985      115    $   450.00
toyota     rav4        2002      180    $   750.00
chevrolet  impala      1985       85    $  1550.00
```

```
$ ./list_cars
Price range (for example, 5000 7500):15000 100000

                                Miles
Make       Model       Year    (000)         Price
--------------------------------------------------
ford       taurus      2004       10    $ 17000.00
```

optional

Advanced gawk Programming

This section discusses some of the advanced features of AWK. It covers how to control input using the **getline** statement, how to use a coprocess to exchange information between gawk and a program running in the background, and how to use a coprocess to exchange data over a network. Coprocesses are available under gawk only; they are not available under awk and mawk.

getline: Controlling Input

Using the **getline** statement gives you more control over the data gawk reads than other methods of input do. When you provide a variable name as an argument to **getline**, **getline** reads data into that variable. The **BEGIN** block of the **g1** program uses **getline** to read one line into the variable **aa** from standard input:

```
$ cat g1
BEGIN   {
        getline aa
        print aa
        }
$ echo aaaa | gawk -f g1
aaaa
```

The next few examples use the **alpha** file:

```
$ cat alpha
aaaaaaaaa
bbbbbbbbb
ccccccccc
ddddddddd
```

Even when **g1** is given more than one line of input, it processes only the first line:

```
$ gawk -f g1 < alpha
aaaaaaaaa
```

When **getline** is not given an argument, it reads input into **$0** and modifies the field variables (**$1**, **$2**, . . .):

```
$ gawk 'BEGIN {getline;print $1}' < alpha
aaaaaaaaa
```

The **g2** program uses a **while** loop in the **BEGIN** block to loop over the lines in standard input. The **getline** statement reads each line into **holdme** and **print** outputs each value of **holdme**.

```
$ cat g2
BEGIN   {
        while (getline holdme)
            print holdme
        }
$ gawk -f g2 < alpha
aaaaaaaaa
bbbbbbbbb
ccccccccc
ddddddddd
```

The **g3** program demonstrates that gawk automatically reads each line of input into **$0** when it has statements in its body (and not just a **BEGIN** block). This program outputs the record number (**NR**), the string **$0:**, and the value of **$0** (the current record) for each line of input.

```
$ cat g3
        {print NR, "$0:", $0}

$ gawk -f g3 < alpha
1 $0: aaaaaaaaa
2 $0: bbbbbbbbb
3 $0: ccccccccc
4 $0: ddddddddd
```

Next **g4** demonstrates that **getline** works independently of gawk's automatic reads and **$0**. When **getline** reads data into a variable, it does not modify either **$0** or any of the fields in the current record (**$1**, **$2**, . . .). The first statement in **g4**, which is the same as the statement in **g3**, outputs the line that gawk has automatically read. The **getline** statement reads the next line of input into the variable named **aa**. The third statement outputs the record number, the string **aa:**, and the value of **aa**. The output from **g4** shows that **getline** processes records independently of gawk's automatic reads.

```
$ cat g4
        {
        print NR, "$0:", $0
        getline aa
        print NR, "aa:", aa
        }

$ gawk -f g4 < alpha
1 $0: aaaaaaaaa
2 aa: bbbbbbbbb
3 $0: ccccccccc
4 aa: ddddddddd
```

The **g5** program outputs each line of input except for those lines that begin with the letter **b**. The first **print** statement outputs each line that gawk reads automatically. Next the **/^b/** ***pattern*** selects all lines that begin with **b** for special processing. The ***action*** uses **getline** to read the next line of input into the variable **hold**, outputs the string **skip this line:** followed by the value of **hold**, and outputs the value of **$1**. The **$1** holds the value of the first field of the record that gawk read automatically, not the record read by **getline**. The final statement displays a string and the value of **NR**, the current record number. Even though **getline** does not change **$0** when it reads data into a variable, gawk increments **NR**.

```
$ cat g5
        # print all lines except those read with getline
        {print "line #", NR, $0}

# if line begins with "b" process it specially
/^b/    {
        # use getline to read the next line into variable named hold
        getline hold

        # print value of hold
        print "skip this line:", hold

        # $0 is not affected when getline reads data into a variable
        # $1 still holds previous value
        print "previous line began with:", $1
        }

        {
        print ">>>> finished processing line #", NR
        print ""
        }
$ gawk -f g5 < alpha
line # 1 aaaaaaaaa
>>>> finished processing line # 1

line # 2 bbbbbbbbb
skip this line: ccccccccc
previous line began with: bbbbbbbbb
>>>> finished processing line # 3

line # 4 ddddddddd
>>>> finished processing line # 4
```

Coprocess: Two-Way I/O

A *coprocess* is a process that runs in parallel with another process. Starting with version 3.1, gawk can invoke a coprocess to exchange information directly with a background process. A coprocess can be useful when you are working in a client/server environment, setting up an *SQL* (page 980) front end/back end, or exchanging data with a remote system over a network. The gawk syntax identifies a coprocess by preceding the name of the program that starts the background process with a |& operator.

Only gawk supports coprocesses

tip The awk and mawk utilities do not support coprocesses. Only gawk supports coprocesses.

The coprocess command must be a filter (i.e., it reads from standard input and writes to standard output) and must flush its output whenever it has a complete line rather than accumulating lines for subsequent output. When a command is invoked as a coprocess, it is connected via a two-way pipe to a gawk program so you can read from and write to the coprocess.

to_upper When used alone the tr utility (page 864) does not flush its output after each line. The **to_upper** shell script is a wrapper for tr that does flush its output; this filter can be run as a coprocess. For each line read, **to_upper** writes the line, translated to uppercase, to standard output. Remove the # before **set –x** if you want **to_upper** to display debugging output.

```
$ cat to_upper
#!/bin/bash
#set -x
while read arg
do
    echo "$arg" | tr '[a-z]' '[A-Z]'
done

$ echo abcdef | ./to_upper
ABCDEF
```

The **g6** program invokes **to_upper** as a coprocess. This gawk program reads standard input or a file specified on the command line, translates the input to uppercase, and writes the translated data to standard output.

```
$ cat g6
    {
    print $0 |& "to_upper"
    "to_upper" |& getline hold
    print hold
    }

$ gawk -f g6 < alpha
AAAAAAAAA
BBBBBBBBB
CCCCCCCCC
DDDDDDDDD
```

The **g6** program has one compound statement, enclosed within braces, comprising three statements. Because there is no ***pattern***, gawk executes the compound statement once for each line of input.

In the first statement, **print $0** sends the current record to standard output. The |& operator redirects standard output to the program named **to_upper**, which is running as a coprocess. The quotation marks around the name of the program are required. The second statement redirects standard output from **to_upper** to a **getline** statement, which copies its standard input to the variable named **hold**. The third statement, **print hold**, sends the contents of the **hold** variable to standard output.

Getting Input from a Network

Building on the concept of a coprocess, gawk can exchange information with a process on another system via an IP network connection. When you specify one of the special filenames that begins with **/inet/**, gawk processes the request using a network connection. The format of these special filenames is

*/inet/**protocol**/**local-port**/**remote-host**/**remote-port***

where ***protocol*** is usually **tcp** but can be **udp**, ***local-port*** is 0 (zero) if you want gawk to pick a port (otherwise it is the number of the port you want to use), ***remote-host*** is the *IP address* (page 960) or *fully qualified domain name* (page 955) of the remote host, and ***remote-port*** is the port number on the remote host. Instead of a port number in ***local-port*** and ***remote-port***, you can specify a service name such as **http** or **ftp**.

The **g7** program reads the **rfc-retrieval.txt** file from the server at **www.rfc-editor.org**. On **www.rfc-editor.org** the file is located at **/rfc/rfc-retrieval.txt**. The first statement in **g7** assigns the special filename to the **server** variable. The filename specifies a TCP connection, allows the local system to select an appropriate port, and connects to **www.rfc-editor.org** on port 80. You can use **http** in place of **80** to specify the standard HTTP port.

The second statement uses a coprocess to send a **GET** request to the remote server. This request includes the pathname of the file gawk is requesting. A **while** loop uses a coprocess to redirect lines from the server to **getline**. Because **getline** has no variable name as an argument, it saves its input in the current record buffer **$0**. The final **print** statement sends each record to standard output. Experiment with this script, replacing the final **print** statement with gawk statements that process the file.

```
$ cat g7
BEGIN           {
    # set variable named server
    # to special networking filename
    server = "/inet/tcp/0/www.rfc-editor.org/80"

    # use coprocess to send GET request to remote server
    print "GET /rfc/rfc-retrieval.txt" |& server

    # while loop uses coprocess to redirect
    # output from server to getline
    while (server |& getline)
        print $0
    }
```

```
$ gawk -f g7

                   Where and how to get new RFCs
                   =============================

RFCs may be obtained via FTP or HTTP or email from many RFC repositories.
The official repository for RFCs is:

        http://www.rfc-editor.org/
...
```

Chapter Summary

AWK is a pattern-scanning and processing language that searches one or more files for records (usually lines) that match specified patterns. It processes lines by performing actions, such as writing the record to standard output or incrementing a counter, each time it finds a match. AWK has several implementations, including awk, gawk, and mawk.

An AWK program consists of one or more lines containing a ***pattern*** and/or ***action*** in the following format:

> *pattern { action }*

The ***pattern*** selects lines from the input. An AWK program performs the ***action*** on all lines that the ***pattern*** selects. If a ***program*** line does not contain a ***pattern,*** AWK selects all lines in the input. If a program line does not contain an ***action,*** AWK copies the selected lines to standard output.

An AWK program can use variables, functions, arithmetic operators, associative arrays, control statements, and C's **printf** statement. Advanced AWK programming takes advantage of **getline** statements to fine-tune input, coprocesses to enable gawk to exchange data with other programs (gawk only), and network connections to exchange data with programs running on remote systems on a network (gawk only).

Exercises

1. Write an AWK program that numbers each line in a file and sends its output to standard output.
2. Write an AWK program that displays the number of characters in the first field followed by the first field and sends its output to standard output.

3. Write an AWK program that uses the **cars** file (page 541), displays all cars priced at more than $5,000, and sends its output to standard output.
4. Use AWK to determine how many lines in **/usr/share/dict/words** contain the string **abul**. Verify your answer using grep.

Advanced Exercises

5. Experiment with pgawk (available only with gawk). What does it do? How can it be useful?
6. Write a gawk (not awk or mawk) program named **net_list** that reads from the **rfc-retrieval.txt** file on **www.rfc-editor.org** (see "Getting Input from a Network" on page 562) and displays a the last word on each line in all uppercase letters.
7. Expand the **net_list** program developed in Exercise 6 to use **to_upper** (page 561) as a coprocess to display the list of cars with only the make of the cars in uppercase. The model and subsequent fields on each line should appear as they do in the **cars** file.
8. How can you get gawk (not awk or mawk) to neatly format—that is, "pretty print"—a gawk program file? (*Hint:* See the gawk man page.)