4. awk PROGRAMMING LANGUAGE

Introduction

Suppose you want to tabulate some survey results stored in a file; print various reports summarizing the results; generate form letters; reformat a data file for one application package to use with another package; count the occurrences of particular strings in a file; or globally substitute a string in many files without ever invoking an editor. The tasks common to all these examples are retrieving and processing data in files. You can simplify such tasks with **awk**. **awk** is an interpretive programming language designed to handle these tasks. It is also the name of the operating system command you use to run **awk** programs. This tutorial describes the language and command.

As the examples above suggest, **awk** is suited for various applications. It is designed to retrieve and manipulate data efficiently from any files containing mixtures of words and/or numbers. Although **awk** has been most commonly used to generate reports, it has also been used to choose names randomly, to sort error messages in a programmer's manual, and even to implement database systems and compilers.

Why should you choose **awk** instead of another language to implement these tasks? Because **awk** is a relatively easy language to learn. Its simple but powerful syntax can be summarized in a few pages. For instance, unlike C, Pascal, and some other languages, **awk** does not require that you explicitly initialize variables in programs. However, it provides the same powerful flow-control statements (like **for** and **if-else**) provided by these other languages. **awk** can also save you some time, because you don't need to compile **awk** programs before running them. This makes it a good language for prototyping.

This chapter is intended to make it easy for you to start writing and running your own **awk** programs. The chapter begins with the basics of **awk**. The rest of the chapter describes the complete language and is somewhat less tutorial. For the experienced **awk** user, there's a summary of the language at the end of the chapter.

To use this chapter, you should be familiar with the operating system and shell programming. Although you don't need other programming experience, some knowledge of C is beneficial because many constructs found in **awk** also occur in C.

Basic awk

This section provides enough information for you to write and run some of your own programs. Each topic presented below is discussed in more detail in later sections.

Program Structure

The basic operation of **awk**(1) is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **awk** program is a sequence of pattern-action statements, as Figure 4-1 shows.

Structure:

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

Example:

```
$1 == "address"   { print $2, $3 }
```

Figure 4-1. awk Program Structure and Example

The example in the figure is a typical **awk** program, consisting of one pattern-action statement. Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts over. This process typically continues until all the input has been read.

Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in:

```
$1 == "name"
```

the matching line is printed. If there is no pattern with an action, as in:

```
{ print $1, $2 }
```

then the action is performed for every input line. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Usage

There are two ways to run an **awk** program. First, you can type the command line:

```
awk 'pattern-action statements'  optional list of input files
```

to execute the pattern-action statements on the set of named input files. For example, you could say:

```
awk '{ print $1, $2 }' file1 file2
```

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like **$** from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk**(1) reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (–) as one of the input files. For example,

```
awk '{ print $3, $4 }' file1 –
```

says to read input first from **file1** and then from the standard input.

The arrangement above is convenient when the **awk** program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file and use the **–f** option to fetch it:

```
awk –f program file  optional list of input files
```

For example, the following command line says to fetch **myprogram** and read from the file **file1**:

```
awk –f myprogram file1
```

Fields

An **awk**(1) program normally reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline. The **awk** program then splits each record into fields, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the **awk** programs in this chapter, we use the following file, **countries**. Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent where it is, for the ten largest countries in the world. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.) The wide space between fields is a tab in the original input; a single blank separates `North` and `South` from `America` .

```
USSR        8650    262     Asia
Canada      3852    24      North America
China       3692    866     Asia
USA         3615    219     North America
Brazil      3286    116     South America
Australia   2968    14      Australia
India       1269    637     Asia
Argentina   1072    26      South America
Sudan       968     19      Africa
Algeria     920     18      Africa
```

Figure 4-2. The Sample Input File **countries**

This file is typical of the kind of data **awk** is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, in which case the first record of **countries** would have four fields, the second five, and so on. It's possible to set the field separator to just tab, so each line would have four fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default: fields separated by blanks and/or tabs.

The first field within a line is called **$1,** the second **$2,** and so forth. An entire record is called **$0**.

Printing

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line; you can accomplish this with an **awk** program consisting of a single **print** statement:

```
{ print }
```

So the command line:

```
awk '{ print }' countries
```

prints each line of **countries,** copying the file to the standard output. The **print** statement can also be used to print parts of a record. The following program, for instance, prints the first and third fields of each record:

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

Thus, the program:

awk '{ print $1, $3 }' countries

produces as output the sequence of lines:

```
USSR 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

When printed, items separated by a comma in the **print** statement are separated by the output field separator, which is by default a single blank. Each line printed is terminated by the output record separator, which is by default a newline.

NOTE

The remainder of this chapter shows only **awk** programs, without the command line that invokes them. Each complete program can be run either by enclosing it in quotes as the first argument of the **awk** command or by putting it in a file and invoking **awk** with the **–f** flag, as discussed in "**awk** Command Usage." If no input is mentioned in an example, the input is assumed to be the file **countries**.

Formatted Printing

For more carefully formatted output, **awk** provides a C-like **printf** statement:

printf *format*, $expr_1$, $expr_2$, . . ., $expr_n$

The **printf** statement prints the $expr_i$'s according to the specification in the string *format*. For example, the **awk** program:

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field (`$1`) as a string of 10 characters (right justified), then a space, then the third field (`$3`) as a decimal number in a six-character field, then a newline (\n). With input from the file **countries,** this program prints an aligned table:

```
      USSR   262
    Canada    24
     China   866
       USA   219
    Brazil   116
 Australia    14
     India   637
 Argentina    26
     Sudan    19
   Algeria    18
```

With **printf,** no output separators or newlines are produced automatically; you must create them yourself. That is the purpose of the **\n** in the format specification. A full description of **printf** can be found under "The **printf** Statement" in this chapter.

Simple Patterns

You can select specific records for printing or other processing with simple patterns. The **awk** language has three kinds of patterns that you can use.

First, you can use patterns called "relational expressions" that make comparisons. For example, the operator == tests for equality. To print the lines in which the fourth field equals the string `Asia`, we can use a program consisting of the single pattern:

```
$4 == "Asia"
```

With the file **countries** as input, this program yields:

```
USSR       8650    262    Asia
China      3692    866    Asia
India      1269    637    Asia
```

The complete set of comparisons is >, >=, <, <=, == (equal to) and != (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with more than 100 million population. The program:

```
$3 > 100
```

is all that is needed. (Remember that the third field in the file **countries** is the population in millions; the program prints all lines in which the third field exceeds 100.)

Second, you can use patterns called "regular expressions" that select records by searching for specified characters. The simplest form of a regular expression is a string of characters enclosed in slashes, such as:

```
/US/
```

This program prints each line that contains the (adjacent) letters US anywhere. With the file **countries** as input, it prints:

```
USSR        8650    262     Asia
USA         3615    219     North America
```

We will have a lot more to say about regular expressions later in this chapter.

Third, you can use two special patterns, **BEGIN** and **END**, that match before the first record has been read and after the last record has been processed. This program uses **BEGIN** to print a title:

```
BEGIN       { print "Countries of Asia:" }
/Asia/      { print "     ", $1 }
```

The output is:

```
Countries of Asia:
     USSR
     China
     India
```

Simple Actions

We have already seen the simplest action of an **awk** program: printing each input line. Now let's consider how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in Variables

Besides reading the input and splitting it into fields, **awk**(1) counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields. So the program:

```
{ print NR, NF }
```

prints the number of each line and how many fields it has, while

```
{ print NR, $0 }
```

prints each record preceded by its record number.

User-defined Variables

In addition to providing built-in variables like **NF** and **NR, awk** lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file **countries**:

```
        { sum = sum + $3 }
END     { print "Total population is", sum, "million"
          print "Average population of", NR, "countries is", sum/NR }
```

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million
Average population of 10 countries is 220.1
```

Functions

The **awk** language has built-in functions that handle common arithmetic and string operations for you. For example, there's an arithmetic function that computes square roots. There is also a string function that substitutes one string for another. The **awk** language also lets you define your own functions. Functions are described in detail under "Actions" in this chapter.

Useful One-Line Programs

Although **awk** can be used to write large programs of some complexity, many programs are not much more complicated than what we've seen so far. Here is a collection of other short programs that you might find useful and instructive. Most of the program constructs have been introduced earlier in this chapter. They are not explained here, but any new constructs do appear later in this chapter.

Print last field of each input line:
```
{ print $NF }
```
Print 10th input line:
```
NR == 10
```
Print last input line:
```
{ line = $0}
END{ print line }
```
Print input lines that don't have 4 fields:
```
NF != 4 { print $0, " does not have 4 fields" }
```
Print input lines with more than 4 fields:
```
NF > 4
```
Print input lines with last field more than 4:
```
$NF > 4
```
Print total number of input lines:
```
END{ print NR }
```
Print total number of fields:
```
{ nf = nf + NF }
END{ print nf }
```
Print total number of input characters:
```
{ nc = nc + length($0) }
END{ print nc + NR }
```
(Adding **NR** includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:
```
/Asia/{ nlines++ }
END{ print nlines }
```
(The statement `nlines++` has the same effect as `nlines = nlines + 1`.)

Error Messages

If you make an error in an **awk** program, you generally get messages like:

```
awk: syntax error near source line 2
awk: bailing out near source line 2
```

The first message means that you have made a grammatical error that was finally detected near the line specified. The second message means that because of the syntax errors **awk**(1) made no attempt to execute your program.

Sometimes you get a little more help about what the error is, such as a report of missing braces or unbalanced parentheses. For example, running the program:

```
$3 < 200 { print $1, $3
```

that is missing a closing brace, generates the error messages:

```
awk: syntax error near line 2
awk: illegal statement near line 2
```

Some errors may be detected while your program is running. For example, if you try to divide a number by zero, **awk** stops processing and reports the input record number (**NR**) and the line number in the program.

Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that may be used as patterns.

BEGIN and END

BEGIN and **END** are two special patterns that give you a way to control initialization and wrap-up in an **awk** program. **BEGIN** matches before the first input record is read, so any statements in the action part of a **BEGIN** are done before the **awk** command starts to read its first input record. The pattern **END** matches the end of the input, after the last record has been processed.

The following **awk** program uses **BEGIN** to set the field separator to tab (**\t**) and to put column headings on the output. The field separator is stored in a built-in variable called **FS**. Although **FS** can be reset at any time, usually the only sensible place is in a **BEGIN** section, before any input has been read. The program's second **printf** statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The **END** action prints the totals. (Notice that a long line can be continued after a comma.)

```
BEGIN { FS = "\t"
        printf "%10s %6s %5s   %s\n",
      "COUNTRY", "AREA", "POP", "CONTINENT" }
      { printf "%10s %6d %5d   %s\n", $1, $2, $3, $4
      area = area + $2; pop = pop + $3 }
END   { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file **countries** as input, this program produces:

```
   COUNTRY   AREA   POP   CONTINENT
      USSR   8650   262   Asia
    Canada   3852    24   North America
     China   3692   866   Asia
       USA   3615   219   North America
    Brazil   3286   116   South America
 Australia   2968    14   Australia
     India   1269   637   Asia
 Argentina   1072    26   South America
     Sudan    968    19   Africa
   Algeria    920    18   Africa

     TOTAL  30292  2201
```

Relational Expressions

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. The **awk** language has six relational operators (as well as two regular expression matching operators, ˜ (tilde) and **!˜**, which are discussed in the next section) for making comparisons. Figure 4-3 shows these operators and their meanings.

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matches
!~	does not match

Figure 4-3. awk Comparison Operators

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **awk** can tell what is intended. Refer to "Number or String?" in this chapter.) Thus, the pattern **$3>100** selects lines where the third field exceeds 100, and the program:

```
$1 >= "S"
```

selects lines that begin with the letters S, T, U, through Z, which are:

```
USA      3615  219   North America
Sudan    968   19    Africa
```

In the absence of any other information, **awk** treats fields as strings, so the program:

```
$1 == $4
```

compares the first and fourth fields as strings of characters, and with the file **countries** as input, prints the single line for which this test succeeds:

```
Australia   2968   14   Australia
```

If both fields appear to be numbers, the comparisons are done numerically.

Regular Expressions

The **awk** language provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns, called regular expressions, are like those in **egrep**(1) and **lex**(1). The simplest regular expression is a string of characters enclosed in slashes, like:

```
/Asia/
```

This program prints all input records that contain any occurrence of `Asia`. (If a record contains **Asia** as part of a larger string like **Asian** or **Pan-Asiatic**, it is also printed.) In general, if *re* is a regular expression, then the pattern:

/re/

matches any line that contains a substring specified by the regular expression *re*.

To restrict a match to a specific field, you use the matching operators ~ (for matches) and !~ (for does not match). The program:

```
$4 ~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field matches **Asia**, while the program:

```
$4 !~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field does not match **Asia** .

In regular expressions, the symbols:

```
\ ^ $ . [] * + ? () |
```

are metacharacters with special meanings like the metacharacters in the shell. For example, the metacharacters ^ and **$** respectively match the beginning and end of a string, and the metacharacter **.** matches any single character. Thus, the program:

```
/^.$/
```

matches all records that contain exactly one character.

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, **/[ABC]/** matches records containing any one of **A, B** or **C** anywhere. Ranges of letters or digits can be abbreviated within brackets: **/[a-zA-Z]/** matches any single letter.

If the first character after the **[** is a **^**, this complements the class so it matches any character not in the set: **/[^a-zA-Z]/** matches any non-letter. The program:

```
$2 !~ /^[0-9]+$/
```

prints all records in which the second field is not a string of one or more digits (^ for beginning of string, `[0-9]+` for one or more digits, and **$** for end of string). Programs of this nature are often used for data validation.

Parentheses **()** are used for grouping and the pipe symbol | is used for alternatives. The program:

```
/(apple|cherry) (pie|tart)/
```

matches lines containing any one of the four substrings **apple pie, apple tart, cherry pie,** or **cherry tart** .

To turn off the special meaning of a metacharacter, precede it by a \ (backslash). Thus, the program:

```
    /a\$/
```

prints all lines containing **a** followed by a dollar sign.

In addition to recognizing metacharacters, the **awk** command recognizes the following C escape sequences within regular expressions and strings:

\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\ddd	octal value *ddd*
\\"	quotation mark
\c	any other character *c* literally

For example, to print all lines containing a tab, use the program:

```
    /\t/
```

awk interprets any string or variable on the right side of a ~ or !~ as a regular expression. For example, we could have written the program:

```
    $2 !~ /^[0-9]+$/
```

as:

```
    BEGIN     { digits = "^[0-9]+$" }
    $2 !~ digits
```

When a literal quoted string like "^**[0-9]+$** is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. The reason may seem arcane, but it is merely that one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose we want to match strings containing the letter A followed by a dollar sign. The regular expression for this pattern is **a\$**. If we want to create a string to represent this regular expression, we must add one more backslash: "**a\\$**". The regular expressions on each of the following lines are equivalent.

```
    x ~ "a\\$"         x ~ /a\$/
    x ~ "a\$"          x ~ /a$/
    x ~ "a$"           x ~ /a$/
    x ~ "\\t"          x ~ /\t/
```

Of course, if the context of a matching operator is:

```
    x ~ $1
```

then the additional level of backslashes is not needed in the first field.

The precise form of regular expressions and the substrings they match is given in Figure 4-4. The unary operators *, +, and ? have the highest precedence, then concatenation, and then alternation |. All operators are left associative.

Expression	Matches
c	any non-metacharacter *c*
\c	character *c* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^*s*]	any character not in set *s*
*r**	zero or more *r*'s
r+	one or more *r*'s
r?	zero or one *r*
(*r*)	*r*
r_1r_2	r_1 then r_2 (concatenation)
$r_1 \| r_2$	r_1 or r_2 (alternation)

Figure 4-4. awk Regular Expressions

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators || (or), && (and), and ! (not). For example, suppose we want to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is `Asia` and the third field exceeds 500:

```
    $4 == "Asia" && $3 > 500
```

The following program selects lines with `Asia` or `Africa` as the fourth field.

```
$4 == "Asia" || $4 == "Africa"
```

Another way to write the latter query is to use a regular expression with the alternation operator |:

```
$4 ~ /^(Asia|Africa)$/
```

The negation operator **!** has the highest precedence, then **&&**, and finally ||. The operators **&&** and || evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in:

pat_1, pat_2 { ... }

In this case, the action is performed for each line between an occurrence of pat_1 and the next occurrence of pat_2 (inclusive). As an example, the pattern:

```
/Canada/, /Brazil/
```

matches lines starting with the first line that contains the string Canada up through the next occurrence of the string Brazil :

```
Canada      3852    24     North America
China       3692    866    Asia
USA         3615    219    North America
Brazil      3286    116    South America
```

Similarly, since **FNR** is the number of the current record in the current input file (and **FILENAME** is the name of the current input file), the program:

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

prints the first five records of each input file with the name of the current input file prepended.

Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they may also be a combination of one or more statements. This section describes the statements that can make up actions.

Built-in Variables

Figure 4-5 lists the built-in variables that **awk** maintains. Some of these we have already met; others are used in this and later sections.

Variable	Meaning	Default
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	`%.6g`
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	set by **match()**	-
RLENGTH	set by **match()**	-

Figure 4-5. awk Built-in Variables

Arithmetic

Actions use conventional arithmetic expressions to compute numeric values. As a simple example, suppose we want to print the population density for each country in the file **countries**. Since the second field is the area in thousands of square miles and the third field is the population in millions, the expression **1000 * $3 / $2** gives the population density in people per square mile. This expression can be an element of a program such as:

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

Applied to the file **countries**, the program prints the name of each country and its population density.

```
         USSR   30.3
       Canada    6.2
        China  234.6
          USA   60.6
       Brazil   35.3
    Australia    4.7
        India  502.0
    Argentina   24.3
        Sudan   19.6
      Algeria   19.6
```

Arithmetic is done internally in floating-point format. The arithmetic operators are +, −, *, /, % (remainder) and ^ (exponentiation; ** is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **awk** recognizes and produces scientific (exponential) notation: `1e6`, `1E6`, `10e5`, and `1000000` are numerically equal.

The **awk** language has assignment statements like those found in C. The simplest form is the assignment statement

$v = e$

where v is a variable or field name, and e is an expression. For example, to compute the total population and number of Asian countries, we could write:

```
$4 == "Asia"      { pop = pop + $3; n = n + 1 }
END               { print "population of", n,\
                  "Asian countries in millions is", pop }
```

(A long **awk** statement can also be split across several lines by continuing each line with a \, as in the **END** action shown here. Applied to **countries,** this program produces:

```
     population of 3 Asian countries in millions is 1765
```

The action associated with the pattern `$4 == "Asia"` contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, yet everything works properly because **awk** initializes each variable with the string value "" and the numeric value **0**.

The assignments in the previous program can be written more concisely using the operators += and ++:

```
     $4 == "Asia"  { pop += $3; ++n }
```

The operator += is borrowed from C. It has the same effect as the longer version (the variable on the left is incremented by the value of the expression on the right), but += is shorter and runs faster. The same is true of the ++ operator, which adds 1 to a variable.

The abbreviated assignment operators are +=, -=, *=, /=, %=, and ^=. Their meanings are similar; the expression:

v op= e

has the same effect as:

v = v op e.

The increment operators are ++ and --. As in C, they may be used as prefix operators (**++x**) or postfix (**x++**). If **x** is 1, then **i=++x** increments **x,** then sets **i** to 2, while **i=x++** sets **i** to 1, then increments **x**. An analogous interpretation applies to prefix and postfix --.

Assignment and increment and decrement operators may all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3    { maxpop = $3; country = $1 }
END            { print country, maxpop }
```

Note, however, that this program would not be correct if all values of **$3** were negative.

awk provides the built-in arithmetic functions shown in Figure 4-6.

Function	Value Returned
atan2(*y*,*x*)	arctangent of y/x in the range $-\pi$ to π
cos(*x*)	cosine of *x*, with *x* in radians
exp(*x*)	exponential function of *x*
int(*x*)	integer part of *x* truncated towards 0
log(*x*)	natural logarithm of *x*
rand()	random number between 0 and 1
sin(*x*)	sine of *x*, with *x* in radians
sqrt(*x*)	square root of *x*
srand(*x*)	*x* is new seed for rand()

Figure 4-6. awk Built-in Arithmetic Functions

x and *y* are arbitrary expressions. The function **rand()** returns a pseudo-random floating point number in the range (0,1), and **srand(*x*)** can be used to set the seed of the generator. If **srand()** has no argument, the seed is derived from the time of day.

Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in "**abc**" or "**hello, everyone**" . String constants may contain the C escape sequences for special characters listed in "Regular Expressions" in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The program:

```
{ print NR ":" $0 }
```

prints each record preceded by its record number and a colon, with no blanks. The three strings representing the record number, the colon, and the record are concatenated and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

The **awk** language provides the built-in string functions shown in Figure 4-7. In this table, *r* represents a regular expression (either as a string or as */r/*), *s* and *t* string expressions, and *n* and *p* integers.

Function	Description
gsub(*r*,*s*)	substitute *s* for *r* globally in current record, return number of substitutions
gsub(*r*,*s*,*t*)	substitute *s* for *r* globally in string *t*, return number of substitutions
index(*s*,*t*)	return position of string *t* in *s*, 0 if not present
length	return length of **$0**
length(*s*)	return length of *s*
match(*s*, *r*)	return the position in *s* where *r* occurs
split(*s*,*a*)	split *s* into array *a* on FS, return number of fields
split(*s*,*a*,*r*)	split *s* into array *a* on *r*, return number of fields
sprintf(*fmt*,*expr-list*)	return *expr-list* formatted according to format string *fmt*
sub(*r*,*s*)	substitute *s* for first *r* in current record, return number of substitutions
sub(*r*,*s*,*t*)	substitute *s* for first *r* in *t*, return number of substitutions
substr(*s*,*p*)	return suffix of *s* starting at position *p*
substr(*s*,*p*,*n*)	return substring of *s* of length *n* starting at position *p*

Figure 4-7. awk Built-in String Functions

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(1) . The function **gsub(*r*,*s*,*t*)** replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in **ed,** leftmost longest matches are used.) It returns the number of substitutions made. The function **gsub(*r*,*s*)** is a synonym for **gsub(*r*,*s*,$0)** . For example, the program:

```
{ gsub(/USA/, "United States"); print }
```

transcribes its input, replacing occurrences of `USA` by `United States`. The **sub** functions are similar, except that they only replace the first matching substring in the target string.

The function **index(*s*,*t*)** returns the leftmost position where the string *t* begins in *s*, or zero if *t* does not occur in *s* . The first character in a string is at position 1. For example, the expression:

```
index("banana", "an")
```

returns 2.

The **length** function returns the number of characters in its argument string; thus, the expression:

```
{ print length($0), $0 }
```

prints each record, preceded by its length. (**$0** does not include the input record separator.) The program:

```
length($1) > max { max = length($1); name = $1 }
END                               { print name }
```

when applied to the file **countries**, prints the longest country name: `Australia`.

The **match(***s*, *r***)** function returns the position in string *s* where regular expression *r* occurs, or 0 if it does not occur. This function makes use of the two built-in variables **RSTART** and **RLENGTH**. **RSTART** is set to the starting position of the string; this is the same value as the returned value. **RLENGTH** is set to the length of the matched string. For example, running the program:

```
{ match($0, "ia")
    if (RSTART != 0) print RSTART, RLENGTH
}
```

produces the following output:

```
17 2
18 2
8 2
4 2
6 2
```

The function **sprintf(***format*, $expr_1$, $expr_2$, ..., $expr_n$**)** returns (without printing) a string containing $expr_1$, $expr_2$, ..., $expr_n$ formatted according to the **printf** specifications in the string *format*. "The **printf** Statement" in this chapter contains a complete specification of the format conventions. The statement:

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to `x` the string produced by formatting the values of `$1` and `$2` as a ten-character string and a decimal number in a field of width at least six; `x` may be used in any subsequent computation.

The function **substr(***s*,*p*,*n***)** returns the substring of *s* that begins at position *p* and is at most *n* characters long. If **substr(***s*,*p***)** is used, the substring goes to the end of *s*; that is, it consists of the suffix of *s* beginning at position *p*. For example, we could abbreviate the country names in **countries** to their first three characters by invoking the program.

```
        { $1 = substr($1, 1, 3); print }
```

on this file to produce:

```
        USS 8650 262 Asia
        Can 3852 24 North America
        Chi 3692 866 Asia
        USA 3615 219 North America
        Bra 3286 116 South America
        Aus 2968 14 Australia
        Ind 1269 637 Asia
        Arg 1072 26 South America
        Sud 968 19 Africa
        Alg 920 18 Africa
```

Note that setting `$1` in the program forces **awk** to recompute **$0** and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on the file **countries**, the program:

```
                        { s = s substr($1, 1, 3) " " }
        END             { print s }
```

prints:

```
        USS Can Chi USA Bra Aus Ind Arg Sud Alg
```

by building `s` up a piece at a time from an initially empty string.

Field Variables

The fields of the current record can be referred to by the field variables **$1, $2, ..., $NF**. Field variables share all the properties of other variables — they may be used in arithmetic or string operations, and may be assigned to. So, for example, you can divide the second field of the file **countries** by 1000 to convert the area from thousands to millions of square miles:

```
        { $2 /= 1000; print }
```

or assign a new string to a field:

```
        BEGIN                       { FS = OFS = "\t" }
        $4 == "North America"       { $4 = "NA" }
        $4 == "South America"       { $4 = "SA" }
                                    { print }
```

The `BEGIN` action in this program resets the input field separator `FS` and the output field separator `OFS` to a tab. Notice that the `print` in the fourth line of the program prints the value of **$0** after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, **$(NF–1)** is the second last field of the current record. The parentheses are needed: the value of **$NF–1** is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, **$(NF+1),** has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file **countries** creates a fifth field giving the population density:

```
BEGIN           { FS = OFS = "\t" }
                { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record, but there is usually an implementation limit of 100 fields per record.

Number or String?

Variables, fields, and expressions can have both a numeric value and a string value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like:

```
pop += $3
```

the `pop` and `$3` must be treated numerically, so their values will be coerced to numeric type if necessary.

In a string context like:

```
print $1 ":" $2
```

the `$1` and `$2` must be strings to be concatenated, so they will be coerced if necessary.

In an assignment *v* = *e* or *v op*= *e*, the type of *v* becomes the type of *e*. In an ambiguous context like:

```
$1 == $2
```

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison `"$1 == $2"` will succeed on any pair of the inputs:

```
1     1.0     +1     0.1e+1     10E-1     1e2     10e1     001
```

but fail on the inputs:

(null)	0
(null)	0.0
0a	0
1e50	1.0e50

There are two idioms for coercing an expression of one type to the other:

number ""	concatenate a null string to a *number* to coerce it to type string
string + 0	add zero to a *string* to coerce it to type numeric

Thus, to force a string comparison between two fields, say:

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion **OFMT**.

Uninitialized variables have numeric value 0 and string value "". Nonexistent fields and fields that are explicitly null have only the string value ""; they are not numeric.

Control Flow Statements

The **awk** language provides **if–else, while, do–while,** and **for** statements as well as statement grouping with braces, as in C.

The **if** statement syntax is:

if (*expression*) $statement_1$ **else** $statement_2$

The *expression* acting as the conditional has no restrictions; it can include any of the following:

- The relational operators <, <=, >, >=, ==, and !=

- The regular expression matching operators ~ and !~
- The logical operators ||, **&&**, and **!**
- Juxtaposition for concatenation
- Parentheses for grouping

In the **if** statement, the *expression* is first evaluated. If it is non-zero and non-null, *statement*$_1$ is executed; otherwise *statement*$_2$ is executed. The **else** part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program from "Arithmetic Functions" with an **if** statement results in:

```
{           if (maxpop < $3) {
            maxpop = $3
            country = $1
            }
}
END         { print country, maxpop }
```

The **while** statement is exactly that of C:

while (*expression*) *statement*

The *expression* is evaluated; if it is non-zero and non-null the *statement* is executed and the *expression* is tested again. The cycle repeats as long as the *expression* is non-zero. For example, to print all input fields one per line,

```
{           i = 1
            while (i <= NF) {
            print $i
            i++
            }
}
```

The **for** statement is like that of C:

for (*expression*$_1$; *expression*; *expression*$_2$) *statement*

It has the same effect as:

```
expression₁
while (expression) {
                statement
                expression₂
}
```

so the statement:

```
{ for (i = 1; i <= NF; i++)   print $i }
```

does the same job as the **while** example above. An alternate version of the **for** statement is described in the next section.

The **do** statement has the form:

do *statement* **while** (*expression*)

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement,* it is always executed at least once.

The **do** statement in the program:

```
{ i = 0;
   do { print $i++ }
      while (i <= NF)
}
```

prints each field in a record in a vertical listing.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin. The **next** statement causes **awk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit** statement causes the program to behave as if the end of the input had occurred; no more input is read, and the **END** action (if one has been specified) is executed. Within the **END** action, the statement:

exit *expr*

causes the program to return the value of *expr* as its exit status. If there is no *expr,* the exit status is zero.

Arrays

The **awk** language provides one-dimensional arrays. Arrays and array elements need not be declared; like variables, they spring into existence by being mentioned. An array subscript may be a number or a string.

As an example of a conventional numeric subscript, the statement:

```
x[NR] = $0
```

assigns the current input line to the NPth element of the array `x` . In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the **awk** program:

```
              { x[NR] = $0 }
END           { ... processing ... }
```

The first action merely records each input line, indexed by line number, in the array `x`; processing is done in the **END** statement.

Array elements may also be named by nonnumeric values. This facility gives **awk** a capability rather like the associative memory of Snobol tables. For example, the following program accumulates the total population of Asia and Africa into the associative array `pop` . The `END` in the program action prints the total population of these two continents.

```
/Asia/     { pop["Asia"] += $3 }
/Africa/   { pop["Africa"] += $3 }
END        { print "Asian population in millions is", pop["Asia"]
             print "African population in millions is", pop["Africa"] }
```

On the file **countries,** this program generates:

```
Asian population in millions is 1765
African population in millions is 37
```

If we had used `pop[Asia]` instead of `pop["Asia"]` In this program, the expression would have used the value of the variable `Asia` as the subscript. Since the variable is uninitialized, the values would have been accumulated in `pop[""]` .

Suppose our task is to determine the total area in each continent of the file **countries**. Any expression can be used as a subscript in an array reference. Thus, the statement:

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the array `area` and accumulates the value of the second field in that entry:

```
BEGIN    { FS = "\t" }
         { area[$4] += $2 }
END      { for (name in area)
         print name, area[name] }
```

Invoked on the file **countries**, this program produces:

```
South America 4358
Africa 1888
Asia 13611
Australia 2968
North America 7467
```

This program uses a form of the **for** statement that iterates over all defined subscripts of an array:

for (*i* **in** *array*) *statement*

The *statement* executes with the variable *i* set in turn to each value of *i* for which *array[i]* has been defined. The loop is executed once for each defined subscript, in a random order. A program does not run properly if *i* is altered during the loop.

The **awk** language does not provide multi-dimensional arrays, so you cannot write **x[i,j]** or **x[i][j]**. You can, however, create your own subscripts by concatenating row and column values with a suitable separator. For example, the segment:

```
for (i = 1; i <= 10; i++)
         for (j = 1; j <= 10; j++)
         arr[i "," j] = ...
```

creates an array whose subscripts have the form `i,j`, such as 1,1 or 1,2. (The comma distinguishes a subscript like 1,12 from one like 11,2.)

You can determine whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in *arr*, as in

```
if ("Africa" in area) ...
```

This condition performs the test without the side effect of creating `area["Africa"]`, which would happen if we used

```
if (area["Africa"] != "") ...
```

Note that neither is a test of whether the array `area` contains an element with value `"Africa"` .

It is also possible to split any string into fields in the elements of an array using the built-in function **split**.

The function:

```
split("s1:s2:s3", a, ":")
```

splits the string `s1:s2:s3` into three fields, using the separator `:` and storing `s1` in `a[1]`, `s2` in `a[2]`, and `s3` in `a[3]` . The number of fields found, here three, is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, **FS** is used as the field separator.

An array element may be deleted with the **delete** statement:

> **delete** *arrayname*[*subscript*]

User-Defined Functions

The **awk** language provides user-defined functions. A function is defined as:

> **func** *name*(*argument-list*) {
>   *statements*
> }

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function, these variables refer to the actual parameters when the function is called. There must be no space between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, to define and test the usual recursive factorial function, you could use:

```
func fact(n) {
        if (n <= 1)
        return 1
        else
        return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are passed by value, however; so the function cannot affect their values outside. Within a function, formal parameters are local variables but all other variables are global. (You can have any number of extra formal parameters that are used purely as local variables; because arrays are passed by reference, however, the local variables can only be scalars.) The **return** statement is optional, but the returned value is undefined if execution falls off the end of the function.

Comments

Comments may be placed in **awk** programs: they begin with the character # and end at the end of the line, as in:

```
print x, y# this is a comment
```

Output

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used to generate simple output; **printf** is used for more carefully formatted output. Like the shell, **awk** lets you redirect output, so that output from **print** and **printf** can be directed to files and pipes. This section describes the use of these two statements.

The print Statement

The statement:

print $expr_1$, $expr_2$, ..., $expr_n$

prints the string value of each expression separated by the output field separator followed by the output record separator. The statement:

```
print
```

is an abbreviation for:

```
print $0 .
```

To print an empty line, use:

```
print "" .
```

Output Separators

The output field separator and record separator are held in the built-in variables **OFS** and **ORS**. Initially, **OFS** is set to a single blank and **ORS** to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN    { OFS = ":"; ORS = "\n\n" }
         { print $1, $2 }
```

Notice that:

```
        { print $1 $2 }
```

prints the first and second fields with no intervening output field separator, because **$1 $2** is a string consisting of the concatenation of the first two fields.

The printf Statement

The **printf** statement in **awk** is the same as that in C, except that the **c** and * format specifiers are not supported. The **printf** statement has the general form:

printf *format*, *expr*$_1$, *expr*$_2$, ..., *expr*$_n$

where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in Figure 4-8. Each specification begins with a **%,** ends with a letter that determines the conversion, and may include:

-	left-justify expression in its field
width	pad field to this width as needed; leading 0 pads with zeros
.prec	maximum string width or digits to right of decimal point

Character	Prints Expression as
d	decimal number
e	[-]d.ddddddE[+-]dd
f	[-]ddd.dddddd
g	e or f conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a %; no argument is converted

Figure 4-8. awk Conversion Characters

Here are some examples of **printf** statements along with the corresponding output:

```
printf "%d", 99/2                  49
printf "%e", 99/2                  4.950000e+01
printf "%f", 99/2                  49.500000
printf "%6.2f", 99/2               49.50
printf "%g", 99/2                  49.5
printf "%o", 99                    143
printf "%06o", 99                  000143
printf "%x", 99                    63
printf "|%s|", "January"           |January|
printf "%d", 99/2                  49
printf "|%10s|", "January"         |   January|
printf "|%-10s|", "January"        |January   |
printf "|%.3s|", "January"         |Jan|
printf "|%10.3s|", "January"       |       Jan|
printf "|%-10.3s|", "January"      |Jan       |
printf "%%"                        %
```

The default output format of numbers is **%.6g**; this can be changed by assigning a new value to **OFMT**. **OFMT** also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

Output into Files

It is possible to print output into files instead of to the standard output by using the > and >> redirection operators. For example, the following program invoked on the file **countries** prints all lines where the population (third field) is bigger than 100 into a file called `bigpop`, and all other lines into `smallpop` :

```
$3 > 100    { print $1, $3 >"bigpop" }
$3 <= 100   { print $1, $3 >"smallpop" }
```

Notice that the filenames have to be quoted; without quotes, `bigpop` and `smallpop` are merely uninitialized variables. If the output files were not literals, they would also have to be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME ) }
```

This is because the > operator has higher precedence than concatenation; without parentheses, the concatenation of `tmp` and `FILENAME` would not work.

NOTE

Files are opened once in an **awk** program; If >> is used instead of > to open a file, output is appended to the file rather than overwriting its original contents. However, when the file is subsequently written to, the two operators function exactly the same.

Output into Pipes

It is also possible to direct printing into a pipe with a command on the other end, instead of a file. The statement:

```
print | "command-line"
```

pipes the output of **print** into the *command-line*.

Although we have shown them here as literal strings enclosed in quotes, the *command-line* and filenames can come from variables, etc., as well.

Suppose we want to create a list of continent-population pairs, sorted alphabetically by continent. The **awk** program below accumulates, in an array named `pop`, the population values in the third field for each of the distinct continent names in the fourth field. The program then prints each continent and its population and pipes this output into the `sort` command.

```
BEGIN   { FS = "\t" }
        { pop[$4] += $3 }
END     { for (c in pop)
          print c ":" pop[c] | "sort" }
```

Invoked on the file **countries,** this program yields:

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all these **print** statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named `sort`), but they are created and opened only once in the entire run. So, in the last example, for all `c` in `pop`, only one sort pipe is open.

However, let's say we use **egrep**(1) and the pipe command:

```
... | ( "egrep " c " file-name" )
              #note the need for parens here since
              # otherwise the | has higher precedence
```

Each iteration of this pipe is associated with a new pipe, because c changes in each iteration. As a result, the program complains that it cannot open egrep *<value of c> file* and stops. The entire pipe name needs to be closed to keep the program from failing. The loop needed is:

```
for ( c in pop ) {
  ... | ( "egrep " c " file-name" )
  close ( "egrep " c " file-name" )
  ... }
```

Given this requirement, your program would be clearer if you assigned the pipe each time to a variable:

```
for ( c in pop ) {
  pipe_cmd = "egrep " c " file-name"
  ... | pipe_cmd
  close ( pipe_cmd  )
  ...
  }
```

There is a limit to the number of files that can be open simultaneously. The statement **close(***file***)** closes a file or pipe; *file* is the string used to create it in the first place, as in:

```
close("sort") .
```

Input

There are several ways to give input to an **awk** program. The most common way is to name on the command line the file that contains the input your program needs. In addition, you can use the **getline** function within a program to read in lines. You can also use command line arguments and pipes to provide input. This section describes each of these methods.

Files and Pipes

The most common way to provide input to an **awk** program is to put the data into a file, say **awkdata,** and then execute:

awk *'program'* **awkdata**

The **awk** program reads its standard input if no filenames are given (see "Usage" in this chapter); thus, a second common arrangement is to have another program pipe its output into **awk**. For example, **egrep**(1) selects input lines containing a specified regular expression, but it can do so faster than **awk** since this is the only thing it does. We could, therefore, invoke the pipe:

```
egrep 'Asia' countries | awk '...'
```

The **egrep** command quickly finds the lines containing **Asia** and passes them on to the **awk** program for subsequent processing.

Input Separators

With the default setting of the field separator **FS**, input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
        field1                        field2
      field1
    field1
```

When the field separator is a tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable **FS**. For example, the statement:

```
BEGIN { FS = "(,[ \\t]*)|([ \\t]+)" } ...'
```

sets the field separator to an optional comma followed by any number of blanks and tabs. **FS** can also be set on the command line with the **–F** argument:

```
awk –F'(,[ \t]*)|([ \t]+)' '...'
```

This statement behaves the same as the previous example. Regular expressions used as field separators do not match null strings.

Multi-Line Records

Records are normally separated by newlines, so that each line is a record, but this too can be changed (though only in a limited way). If the built-in record separator variable **RS** is set to the empty string, as in:

```
BEGIN   { RS = "" }
```

then input records can be several lines long; a sequence of empty lines separates records.

A common way to process multiple-line records is to use:

```
BEGIN   { RS = ""; FS = "\n" }
```

to set the record separator to an empty line and the field separator to a newline. There is a limit, however, on how long a record can be; it is usually about 2500 characters. "The **getline** Function" and "Cooperation with the Shell" in this chapter show other examples of processing multi-line records.

The getline Function

The **awk** language's limited facility for automatically breaking its input into records that are more than one line long is not adequate for some tasks. For example, if records are separated not by blank lines but by something more complicated, merely setting **RS** to null doesn't work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

The function **getline** can be used to read input either from the current input or from a file or pipe, by redirection analogous to **printf**. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF, NR,** and **FNR**. **getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and –1 if some error occurred (such as failure to open a file).

To illustrate, suppose we have input data consisting of multi-line records, each of which begins with a line beginning with `START` and ends with a line beginning with `STOP`. The following **awk** program processes these multi-line records a line at a time, putting the lines of the record into consecutive entries of an array:

```
      f[1] f[2] ... f[nf]
```

Once the line containing `STOP` is encountered, the record can be processed from the data in the array:

```
      /^START/ {
              f[nf=1] = $0
              while (getline && $0 !~ /^STOP/)
              f[++nf] = $0
              # now process the data in f[1]...f[nf]
              ...
      }
```

Notice that this code uses the fact that && evaluates its operands left to right and stops as soon as one is true.

The same job can be done by the following program:

```
/^START/ && nf==0{ f[nf=1] = $0 }
nf > 1            { f[++nf] = $0 }
/^STOP/           { # now process the data in f[1]...f[nf]
                      ...
                      nf = 0
}
```

The statement:

```
getline x
```

reads the next record into the variable `x`. No splitting is done; **NF** is not set. The statement:

```
getline <"file"
```

reads from `file` instead of the current input. It has no effect on **NR** or **FNR,** but field splitting is performed and **NF** is set. The statement:

```
getline x <"file"
```

gets the next record from `file` into `x` ; no splitting is done, and **NF, NR** and **FNR** are untouched.

Note that if you use the statement form:

```
getline x < "file"
```

to construct a file name of more than one literal or variable, the file name needs to be placed in parentheses for correct evaluation:

```
while ( getline x < ( ARGV[1].ARGV[2] ) ) {  ... }
```

Without parentheses, a statement such as:

```
getline x < "tmp".FILENAME
```

sets `x` to read the file `tmp` and not `tmp.`*<value of FILENAME>*. For instance, the statement:

```
while ( getline x < "tmp" "file" ) { ... }
```

loops infinitely because `x` is always set to null.

It is also possible to pipe the output of another command directly into **getline**. For example, the statement:

```
while ("who" | getline)
    n++
```

executes `who` and pipes its output into `getline`. Each iteration of the `while` loop reads one more line and increments the variable `n`, so after the `while` loop terminates, `n` contains a count of the number of users. Similarly, the statement:

```
"date" | getline d
```

pipes the output of `date` into the variable `d`, thus setting `d` to the current date.

Figure 4-9 summarizes the **getline** function.

Form	Sets
getline	**$0, NF, NR, FNR**
getline *var*	*var*, **NR, FNR**
getline <*file*	**$0, NF**
getline *var* <*file*	*var*
cmd | **getline**	**$0, NF**
cmd | **getline** *var*	*var*

Figure 4-9. getline Function

Command-line Arguments

The command-line arguments are available to an **awk** program: the array **ARGV** contains the elements **ARGV[0]**, ..., **ARGV[ARGC-1]**; as in C, **ARGC** is the count. **ARGV[0]** is the name of the program (generally **awk**); the remaining arguments are whatever was provided (excluding the program and any optional arguments). The following command line contains an **awk** program that echoes the arguments that appear after the program name:

```
awk '
BEGIN {
  for (i = 1; i < ARGC; i++)
    printf "%s ", ARGV[i]
  printf "\n"
  exit
}' $*
```

The arguments may be modified or added to; **ARGC** may be altered. As each input file ends, **awk** treats the next non-null element of **ARGV** (up to the current value of **ARGC-1**) as the name of the next input file.

There is one exception to the rule that an argument is a filename. If an argument is of the form:

var=value

then the variable *var* is set to the value *value* as if by assignment. Such an argument is not treated as a filename. If *value* is a string, no quotes are needed.

Using awk With Other Commands and the Shell

An **awk** program gains its greatest power when used with other programs. Here we describe some of the ways in which **awk** programs cooperate with other commands.

The system Function

The built-in function **system**(*command-line*) executes the command *command-line,* which may well be a string computed by, for example, the built-in function **sprintf**. The value returned by **system** is the return status of the command executed.

For example, the program:

```
$1 == "#include"  { gsub(/[<>"]/, "", $2); system("cat " $2) }
```

calls the command `cat` to print the file named in the second field of every input record whose first field is `#include`, after stripping any `<`, `>` or `"` that might be present.

Cooperation with the Shell

In all the examples so far, the **awk** program either was in a file from which it was fetched with the **–f** flag, or it appeared on the command line enclosed in single quotes, as in:

awk '{ print $1 }' ...

Since **awk** uses many of the same characters as the shell does, such as **$** and ", surrounding the **awk** program with single quotes ensures that the shell will pass the entire program unchanged to the **awk** interpreter.

Now, consider writing a command **addr** that will search a file **addresslist** for name, address and telephone information. Suppose that **addresslist** contains names and addresses in which a typical entry is a multi-line record such as:

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

We want to search the address list by issuing commands like:

addr Emlin

That is easily done by a program of the form:

```
awk '
BEGIN           { RS = "" }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called **addr** that contains:

```
awk '
BEGIN           { RS = "" }
/'$1'/
' addresslist
```

The quotes are critical here: the **awk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The `$1` is outside the quotes, visible to the shell, which therefore replaces it by the pattern `Emlin` when the command **addr Emlin** is invoked. The **addr** command can be made executable by changing its mode with the following command: **chmod +x addr.**

A second way to implement **addr** relies on the fact that the shell substitutes for **$** parameters within double quotes:

```
awk "
BEGIN           { RS = \"\" }
/$1/
" addresslist
```

Here we must protect the quotes defining `RS` with backslashes, so that the shell passes them on to **awk,** uninterpreted by the shell. The `$1` is recognized as a parameter, however, so the shell replaces it by the pattern when the command **addr** *pattern* is invoked.

A third way to implement **addr** is to use **ARGV** to pass the regular expression to an **awk** program that explicitly reads through the address list with **getline**:

```
awk '
BEGIN    { RS = ""
           while (getline < "addresslist")
              if ($0 ~ ARGV[1])
                 print $0
           exit
}'
```

All processing is done in the `BEGIN` action.

Notice that any regular expression can be passed to **addr**; in particular, it is possible to retrieve by parts of an address or telephone number as well as by name.

Example Applications

The **awk** language has been used in surprising ways. We have seen **awk** programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **awk** programs are significantly shorter than equivalent programs written in more conventional programming languages such as Pascal or C. In this section, we will present a few more examples to illustrate some additional **awk** programs.

Generating Reports

The **awk** language is especially useful for producing reports that summarize and format information. Suppose we wish to produce a report from the file **countries** in which we list the continents alphabetically, and after each continent its countries in decreasing order of population.

```
Africa:
                    Sudan          19
                    Algeria        18

Asia:
                    China         866
                    India         637
                    USSR          262

Australia:
                    Australia      14

North America:
                    USA           219
                    Canada         24

South America:
                    Brazil        116
                    Argentina      26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, we create a list of continent-country-population triples, in which each field is separated by a colon. This can be done with the following program, **triples,** which uses an array called pop indexed by subscripts of the form "continent:country" to store the population of a given country. The print statement in the **END** section of the program creates the list of continent-country-population triples that are piped to the sort routine:

```
BEGIN   { FS = "\t" }
            { pop[$4 ":" $1] += $3 }
END         { for (cc in pop)
                        print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for sort deserve special mention. The -t: argument tells sort to use : as its field separator. The +0 -1 arguments make the first field the primary sort key. In general, $+i$ $-j$ makes fields $i+1$, $i+2$, ..., j the sort key. If $-j$ is omitted, the fields from $i+1$ to the end of the record are used. The +2nr argument makes the third field, numerically decreasing, the secondary sort key (n is for numeric, r for reverse order). Invoked on the file **countries,** this program produces as output

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form we run it through a second **awk** program **format**:

```
BEGIN       { FS = ":" }
{           if ($1 != prev) {
            print "\n" $1 ":"
            prev = $1
            }
            printf "\t%-10s %6d\n", $2, $3
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The command line:

```
awk –f triples countries | awk –f format
```

gives the desired report. As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **awk**'s and **sort**'s.

As an exercise, add to the population report subtotals for each continent and a grand total.

Additional Examples

1. *Word frequencies.*
   The first example illustrates associative arrays for counting. Suppose we want to count the number of times each word appears in the input, where a word is any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order:

```
      { for (w = 1; w <= NF; w++) count[$w]++ }
END   { for (w in count) print count[w], w | "sort -nr" }
```

   The first statement uses the array `count` to accumulate the number of times each word is used. Once the input has been read, the second `for` loop pipes the final count along with each word into the `sort` command.

2. *Accumulation*.
   Suppose we have two files, **`deposits`** and **`withdrawals`**, of records containing a name field and an amount field. For each name we want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

   ```
   awk '
   FILENAME == "deposits"     { balance[$1] += $2 }
   FILENAME == "withdrawals"  { balance[$1] -= $2 }
   END                        { for (name in balance)
                                        print name, balance[name]
   } ' deposits withdrawals
   ```

   The first statement uses the array `balance` to accumulate the total amount for each name in the file `deposits`. The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name will be created by the second statement. The `END` action prints each name with its net balance.

3. *Random choice.*
   The following function prints (in order) `k` random elements from the first `n` elements of the array `A`. In the program, `k` is the number of entries that still need to be printed, and `n` is the number of elements yet to be examined. The decision of whether to print the *i*th element is determined by the test `rand() < k/n`.

   ```
   func choose(A, k, n) {
             for (i = 1; n > 0; i++)
             if (rand() < k/n--) {
             print A[i]
             k--
             }
             }
   }
   ```

4. *Shell facility*.
   The following **awk** program simulates (crudely) the history facility of the operating system shell. A line containing only = re-executes the last command executed. A line beginning with = *cmd* re-executes the last command whose invocation included the string *cmd*. Otherwise, the current line is executed.

```
$1 == "=" { if (NF == 1)
            system(x[NR] = x[NR-1])
                else
            for (i = NR-1; i > 0; i--)
            if (x[i] ~ $2) {
            system(x[NR] = x[i])
            break
            }
                next }

/./           { system(x[NR] = $0) }
```

5. *Form-letter generation.*
   The following program generates form letters, using a template stored in a file called fform.letter :

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

   and replacement text of this form:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

   The BEGIN action stores the template in the array template ; the remaining action cycles through the input data, using gsub to replace template fields of the form $f2 with the corresponding data fields.

```
BEGIN {     FS = "|"
            while (getline <"form.letter")
            line[++n] = $0
}
{           for (i = 1; i <= n; i++) {
            s = line[i]
            for (j = 1; j <= NF; j++)
            gsub("\\$"j, $j, s)
            print s
            }
}
```

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

awk Summary

Command Line

awk *program filenames*
awk –f *program-file filenames*
awk –F*s* set field separator to string *s*; **–Ft** set separator to tab

Patterns

```
BEGIN
END
```
/regular expression/
relational expression
pattern **&&** *pattern*
pattern || *pattern*
(*pattern*)
!*pattern*
pattern, *pattern*
func *name*(*parameter list*) { *statement* }

Control Flow Statements

if (*expr*) *statement* **[else** *statement***]**
if (*subscript* **in** *array*) *statement* **[else** *statement***]**
while (*expr*) *statement*
for (*expr*; *expr*; *expr*) *statement*
for (*var* **in** *array*) *statement*
do *statement* **while** (*expr*)
break
continue
next
exit [*expr*]
function-name(*expr*, *expr*, ...)
return [*expr*]

Input-Output

close(*filename*)	close file
getline	set **$0** from next input record; set **NF, NR, FNR**
getline <*file*	set **$0** from next record of *file*; set **NF**
getline *var*	set *var* from next input record; set **NR, FNR**

getline *var* <*file*	set *var* from next record of *file*
print	print current record
print *expr-list*	print expressions
print *expr-list* >*file*	print expressions on *file*
printf *fmt*, *expr-list*	format and print
printf *fmt*, *expr-list* >*file*	format and print on *file*
system(*cmd-line*)	execute command *cmd-line*, return status

In **print** and **printf** above, >>*file* appends to the *file*, and | *command* writes on a pipe. Similarly, *command* | **getline** pipes into **getline**. **getline** returns 0 on end of file, and –1 on error.

String Functions

gsub(*r*,*s*,*t*)	substitute string *s* for each substring matching regular expression *r* in string *t*, return number of substitutions; if *t* omitted, use **$0**
index(*s*,*t*)	return index of string *t* in string *s*, or 0 if not present
length(*s*)	return length of string *s*
match(*s*, *r*)	return position in *s* where regular expression *r* occurs, or 0 if *r* is not present
split(*s*,*a*,*r*)	split string *s* into array *a* on regular expression *r*, return number of fields; if *r* omitted, **FS** is used in its place
sprintf(*fmt*, *expr-list*)	print *expr-list* according to *fmt*, return resulting string
sub(*r*,*s*,*t*)	like gsub except only the first matching substring is replaced
substr(*s*,*i*,*n*)	return *n*-char substring of *s* starting at *i*; if *n* omitted, use rest of *s*

Arithmetic Functions

atan2(*y,x*)	arctangent of *y / x* in radians
cos(*expr*)	cosine (angle in radians)
exp(*expr*)	exponential
int(*expr*)	truncate to integer
log(*expr*)	natural logarithm
rand()	random number between 0 and 1
sin(*expr*)	sine (angle in radians)
sqrt(*expr*)	square root
srand(*expr*)	new seed for random number generator; use time of day if no *expr*

Operators (Increasing Precedence)

= += –= *= /= %= ^=	assignment
||	logical OR
&&	logical AND
~ !~	regular expression match, negated match
< <= > >= != ==	relationals
blank	string concatenation
+ –	add, subtract
* / %	multiply, divide, mod
+ – !	unary plus, unary minus, logical negation
^	exponentiation (** is a synonym)
++ —	increment, decrement (prefix and postfix)
$	field

Regular Expressions (Increasing Precedence)

c	matches non-metacharacter *c*
\c	matches literal character *c*
.	matches any character but newline
^	matches beginning of line or string
$	matches end of line or string
[*abc...*]	character class matches any of *abc...*
[^*abc...*]	negated class matches any but *abc...* and newline
r1|*r2*	matches either *r1* or *r2*
r1r2	concatenation: matches *r1*, then *r2*
r+	matches one or more *r*'s
*r**	matches zero or more *r*'s
r?	matches zero or one *r*'s
(*r*)	grouping: matches *r*

Built-In Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0..`ARGC-1`)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)
NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default `%.6g`)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)
RSTART	set by **match()**
RLENGTH	set by **match()**

Limits

Any particular implementation of **awk** enforces some limits. Here are typical values:

100 fields
2500 characters per input record
2500 characters per output record
1024 characters per individual field
1024 characters per printf string
400 characters maximum quoted string
400 characters in character class
15 open files
1 pipe
numbers are limited to what can be represented on the local
 machine, e.g., 1e–38..1e+38

Initialization, Comparison, and Type Coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the assignment:

var = *expr*

its type is set to that of the expression. (Assignment includes +=, –=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in

```
v1 = v2
```

then the type of `v1` becomes that of `v2`.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges, such as:

```
expr + 0
```

and to string by:

```
expr ""
```

(i.e., concatenation with a null string).

Uninitialized variables have the numeric value **0** and the string value "" . Accordingly, if **x** is uninitialized, the statement:

```
if (x) ...
```

is false, and the statements:

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

are all true. But note that the statement:

```
if (x == "0") ...
```

is false.

The type of a field is determined by context when possible; for example, the statement:

```
$1++
```

clearly implies that **$1** is to be numeric, and the statement:

```
$1 = $1 "," $2
```

implies that **$1** and **$2** are both to be strings. Coercion is done as needed.

In contexts where types cannot be reliably determined, such as in the statement:

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value "" ; they are not numeric. Non-existent fields (i.e., fields past **NF**) are treated this way, too.

As it is for fields, so it is for array elements created by **split()**.

Mentioning a variable in an expression causes it to exist, with the value "" as described above. Thus, if `arr[i]` does not currently exist, the statement:

```
        if (arr[i] == "") ...
```

causes it to exist with the value "" so the **if** is satisfied. The special construction:

```
        if (i in arr) ...
```

determines whether `arr[i]` exists, without the side effect of creating it if it does not.