2 Matching Patterns and Processing Information with awk

This chapter describes the awk command, a tool with the ability to match lines of text in a file and a set of commands that you can use to manipulate the matched lines. In addition to matching text with the full set of extended regular expressions described in Chapter 1, awk treats each line, or **record**, as a set of elements, or **fields**, that can be manipulated individually or in combination. Thus, awk can perform more complex operations, such as:

- Writing selected fields of a record
- Reordering or replacing the contents of a record; for example, to change syntax in a program source file or change system calls when porting from one system to another
- Processing input to find numeric counts, sums, or subtotals
- Verifying that a given field contains only numeric information
- Checking to see that delimiters are balanced in a programming file
- Processing data contained in fields within records
- Changing data from one program into a form that can be used by a different program

2.1 Versions of the awk Utility

The DIGITAL UNIX operating system provides several versions of the awk utility:

- The awk command invokes a version that is extended from the original design of Aho, Weinberg, and Kernighan to offer many additional features. This version is XPG4 compliant. Some or all of the extended features might or might not be present in other systems' versions of awk; thus, programs using these features might present portability problems.
- The gawk command invokes an extended version that is similar to awk. This version is provided by the Free Software Foundation.

For information about unique features of gawk, see the gawk(1) reference page.

2.2 Running the awk Program

The awk command has the following syntax:

awk [[-F*ERE*]] [[-v *var*=*val*]] { [-f *prog_file*] | [*prog_text*] } [*file1* [*file2 ...*]]

Table 2-1 describes the flags for the awk command.

Table 2-1: Flags for the awk Command

Flag	Description
-F*ERE*	Specifies an extended regular expression to be used as a field separator. By default, awk uses white space (any number of adjacent tabs or spaces) to separate fields in a record. To specify an alternate separator containing white space or a shell metacharacter, enclose the entire flag in apostrophes. For example: `% awk '-F[Tab]' '-f myprog' report` The -F flag must precede any other command-line argument.
-v *var=val*	Assigns the value *val* to a variable named *var*; such assignments are available to the BEGIN block of a program.
-f *prog_file*	Specifies the name of a file containing an awk program. This flag requires a file name as an argument. The awk command accepts multiple -f flags, concatenating all the program files and treating them as a single program.

You can specify the awk program to be executed either with the -f *prog_file* flag or as a program on the command line. Enclose a command-line program with apostrophes (' ') or quotation marks (" ") as needed to control file name expansion and variable substitution.

Usually, you create an awk program file before running awk. The program file is a series of statements that look like the following:

```
pattern { action }
```

In this structure, a *pattern* is one or more expressions that define the text to be matched. Patterns can consist of the following:

- BEGIN or END
- Boolean combinations of regular expressions using the operators ! (NOT) , || (Logical OR), and && (AND), with parentheses for grouping expressions
- Boolean combinations of relational operations on strings, numbers, fields, and variables
- Ranges of records, specified in this way:

  ```
  pattern1,pattern2
  ```

An *action* is one or more steps to be executed, designated with awk commands, operands, and operators. Actions can consist of the following:

- Assignment statements

- Statements to format and print data
- Tests to alter the flow of control
- Control structures, such as `if-else`, `while`, and `for` statements
- Redirection of output to one or more output streams besides standard output
- Piping of output and input

The braces (`{ }`) are delimiters separating the action from the search pattern. Actions can be specified on a single line, or on multiple lines to give a visual structure to the program. If you place an action consisting of several commands on one line, separate the commands with semicolons (`;`). For example, either of the two following programs will find every record containing either "Gunther" or "gunther". For each matching record, it will print two lines, first the number of the record on which the match was made and then the first two fields of the matched record:

Program 1:

```
/[Gg]unther/ { print "Record:", NR ; print $1, $2 }
```

Program 2:

```
/[Gg]unther/ {
  print "Record:", NR
  print $1, $2
}
```

Output from these programs might look like the following:

```
Record: 382
Schuller Gunther
Record: 397
schwarz gunther
```

Both the pattern and the action are optional elements of a program line. If you omit the pattern, `awk` performs the action on every record in the file; if you omit the action, `awk` copies the record to standard output. A null program passes its input unmodified to the output.

Once you create the program file, enter the `awk` command on the command line as follows:

```
$ awk -f progfile infile > outfile
```

This command uses the program in `progfile` to process `infile`, and writes the output to `outfile`. The input file is not changed.

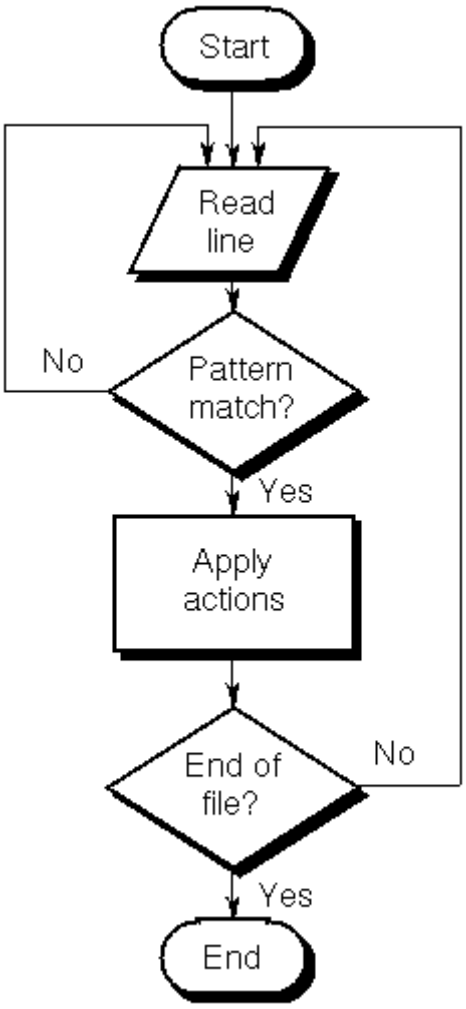
With a short program, you can accomplish the same job by entering the program on the command line before the name of the input file. For example:

```
$ awk '/[Gg]unther/ { print $1, $2 }' infile
```

When you use `awk` in this way, enclose the program in single or double quotation marks as required to control shell file name expansion and variable substitution.

When you start awk, it reads the program, checking for syntax. It then reads the first record of the input file, testing the record against each of the patterns in the program file in order of their appearance. When awk finds a pattern that matches the record, it performs the associated action. Then awk continues to search for matches in the program file. When it has compared the first input record against all patterns in the program file and performed all the actions required for that record, awk reads the next input record and repeats the program with that record. Processing continues in this manner until the end of the input file is reached. Figure 2-1 is a flowchart of this sequence. Compare the operation of awk with the very similar operation of the sed editor, shown in Figure 3-1.

Figure 2-1: Sequence of awk Processing



2.3 Printing in awk

You can use either the print command or the printf command to produce output in awk. The print command prints its arguments as already described; that is, arguments separated by commas are printed separated by the current output field separator, and arguments not separated by commas are concatenated as they are printed.

The printf command has a syntax identical to that of the printf statement in the C programming language:

```
printf( "format", value1 [, value2 , ...] )
```

This command prints the arguments *value1*, *value2*, and so on, formatted as defined by the *format* string. Refer to the awk(1) and printf(3) reference pages for information on constructing format specifiers.

2.4 Using Variables in awk

The awk program uses variables to manipulate information. Variables are of the following three types:

- Simple variables
- Field variables
- Array variables

The awk language supports the set of built-in variables described in Section 2.4.4. You can also create and modify variables of all three types. For example, the following assignment statement creates a variable named var whose value is the sum of the third and fourth field variables in the current record:

```
var = $3 + $4
```

You can use variables as part of a pattern, and you can manipulate them in actions. For example, the following program assigns a value to a variable named tst and then uses tst as part of a pattern for further actions:

```
{ tst = $1 }
tst == $3 { print }
```

Section 2.4.1, Section 2.4.2, and Section 2.4.3 discuss the three types of variables and how to use them. Some of the examples in these sections illustrate the use of other awk features; beginning with Section 2.5, the remaining sections in the chapter provide more detailed information about these features.

2.4.1 Simple Variables

You can create any number of simple (scalar) variables, assigning values to them as required. If you refer to a variable before explicitly assigning a value to it, awk creates the variable and assigns it a value of 0 (zero). Variables can have numeric (floating-point) values or string values depending on their use in the action expression. For example, in the expression `x = 1`, *x* is a numeric variable. Similarly, in the expression `x = "smith"`, *x* is a string variable. However, awk converts freely between strings and numbers when needed. Therefore, in the expression `x = "3"+"4"`, awk assigns a value of 7 (numeric) to *x*, even though the arguments are literal strings. If you use a variable containing a nonnumeric value in a numeric expression, awk assigns it a numeric value of 0. For example:

```
y = 0
z = "ABC"
x = y+z
print x, z
```

This sequence prints "0 0" because *y* is assigned a value of 0 and *z* assumes a value of 0 when used numerically.

You can force a variable to be treated as a string by concatenating the null string (`""`) to the variable; for example, `x = 2 ""`. (See Section 2.13 for information on concatenating strings.) You can force a variable to be treated numerically by adding zero to it. Forcing variables to be treated as particular types can be useful. For example, if *x* is "0100" and *y* is "1", `awk` normally treats both variables as numerics and considers that *x* is greater than *y*. Forcing both variables to be treated as strings causes *x* to be less than *y* because "0" precedes "1" in standard character collating sequences.

2.4.2 Field Variables

Fields in the current record, also called **field variables**, share the properties of simple variables. They can be used in arithmetic or string operations and can be assigned numeric or string values. You can modify `$0` explicitly in `awk`. The following action replaces the first field with the record number and then prints the resulting record:

```
{ $1 = NR; print }
```

The next example adds the second and third fields and stores the result in the first field:

```
{ $1 = $2 + $3; print $0 }
```

(Printing `$0` is identical to printing with no arguments.)

You can use numeric expressions for field references; the following example prints the first, second, and sixth fields:

```
i = 1
n = 5
{ print $i, $(i+1), $(i+n) }
```

As described in Section 2.4.1, `awk` converts between string and numeric values. How you use a field determines whether `awk` treats it as a string or numeric value. If it cannot tell how a given field is used, `awk` treats it as a string.

The `awk` program splits input records into fields as needed. You can split any literal string or string variable into an array by using the `split` function. For example:

```
x = split(s, array1)
y = split("Thu Dec 18 11:19:40 EST 1992", array2)
```

The first line in this example splits the variable *s* into elements of an array named `array1`, creating `array1[1]` to `array1[`*n*`]` where *n* is the number of fields in the string. The second line splits a literal string in the same manner into `array2`. The `split` function can split strings by using an alternative field separator; see Section 2.10 for more information on using this function. See Section 2.4.3 for information on using arrays.

2.4.3 Array Variables

Like field variables, array variables share the properties of simple variables. They can be used in arithmetic or string operations and can be assigned numeric or string values. You do not need to declare or initialize array elements; awk creates them and initializes them to zero upon first reference. Subscripts are indicated by being enclosed in brackets. You can use any value that is not null, including a string value, for a subscript. An example of a numeric subscript follows:

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

This expression creates the *NR*th element of the array *x* and assigns the contents of the current input record to it. The following example illustrates using string subscripts:

```
/apple/  { x["apple"]++ }
/orange/ { x["orange"]++ }
END      { print x["apple"], x["orange"] }
```

For each input record containing apple, this program increments the *apple*th element of array *x* (and similarly for orange), thereby producing and printing a count of the records containing each of these words. (Note that this is not a count of the number of occurrences, because a word can appear more than once in a record.)

Problems can occur when you use an if or while statement to locate an array element. (See Section 2.11 for information on using these and other control structures.) If the array subscript does not exist, the statement adds the subscript as a new B-tree node with a null value. For example:

```
{ if (exists[$2] == 1) }
```

To avoid this type of problem, use code similar to the following, in which *i* is printed after an if statement within a for loop:

```
for (i in exists) {
  if (exists[i] != "") print i
}
```

Also use this type of coding when while is used with a relational operator.

2.4.4 Built-In awk Variables

The awk programs recognize the set of built-in variables listed in Table 2-2.

Table 2-2: Built-In Variables in awk

Variable	Description
$0	The contents of the current record.
$*n*	The contents of field *n* of the input record. In awk you can modify the entire record ($0).
ARGC	A count of the arguments given to awk. This variable is modifiable. Does not include the command name, flags preceded by minus signs, the script file name (if any), or variable assignments.

Variable	Description
ARGV	An array containing the arguments given to awk. The elements of this array are modifiable. Does not include the command name, flags preceded by minus signs, the script file name (if any), or variable assignments.
CONVFMT	The conversion format for numbers (by default, %.6g).
ENVIRON	A modifiable array containing the current set of environment variables; accessible by ENVIRON[*var*], where *var* is the name of the environmental variable. Changing an element in this array does not affect the environment passed to commands that awk spawns by redirection, piping, or the system() function.
FILENAME	The name of the current input file. If no input file was named, FILENAME contains a single minus sign. Inside a BEGIN action, FILENAME is undefined. Inside an END action, FILENAME reflects the last file read.
FNR	The number of the current record within the current file. Differs from NR if multiple files are being processed and the current file is not the first file read.
FS	The character or expression used for a field separator. By default, any amount of white space. In awk, field separators can be multibyte regular expressions and can be multiply defined. For example, the following statement defines either a comma followed by any amount of white space or at least one white-space character as the field separator: FS = ",[**[Tab]**]*|[**[Tab]**]+"
NF	The number of fields in the current record.
NR	The number of the current record, counted sequentially from the beginning of the first file read. Differs from FNR if multiple files are being processed and the current file is not the first file read.
OFMT	The format specification for numbers on output (by default, %.6g).
OFS	The output field separator; or string inserted between fields when the data is written. By default, a space character.
ORS	The character used for the output record separator (the character between records when the data is written). By default, a newline character.
RLENGTH	The length of the string matched by match(); set to -1 if no match.
RS	The character used for a record separator.

Variable	Description
RSTART	The index (position within the string) of the first character matched by `match()`; set to 0 if no match.
SUBSEP	The separator for multiple subscripts in array elements (by default \034, the ASCII FS character).

See the awk(1) reference page for more information about these variables.

2.5 More About Using Regular Expressions as Patterns

The simplest regular expression is a literal string of characters. Expressions in `awk` must be enclosed in slashes. To include a slash as part of an expression, escape the slash with a backslash. For example, `/\/usr\/share/` is an expression that matches the string `/usr/share`.

Following is an example of an `awk` program that prints all records containing the string `the`.

```
/the/
```

Because this expression does not specify blanks or other qualifiers, the program displays records containing "the" as a separate word and records containing the string as part of words such as "northern". Regular expressions are case sensitive. To find either "The" or "the", use a bracketed expression as follows:

```
/[Tt]he/
```

The `awk` language supports the full set of extended regular expressions described in Chapter 1. Additionally, in `awk` the circumflex (`^`) and dollar sign (`$`) can apply to a specific field or variable as well as to the entire line. The following example will match a field consisting of the word "cat" or the word "cats" but will not match any word containing these strings (such as "concatenate"):

```
{ for (i=1;i<=NF;i++) if ($i ~ /^cats?$/) print }
```

2.6 Using Relational Expressions and Combined Expressions as Patterns

Relational expressions lets you restrict a match to a specific field of a record or to make other tests, either numeric or string-related. One example earlier in this chapter (in Section 2.4) illustrates the use of relational expressions in patterns. The `awk` program defines the following relational operators for use in building patterns:

==	Equivalent
!=	Not equivalent

<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
~	Matches regular expression
!~	Does not match regular expression

Use the == (equivalent) and != (not equivalent) operators to test literal strings and numeric values. For example:

```
str == "literal string"
num != 23
$NF == 1991
```

The last line in this example uses the $*n* syntax combined with the built-in variable NF to test the value of the last field of a record. To test against regular expressions, use the ~ (matches regular expression) and !~ (does not match regular expression) operators as follows:

```
str ~ /[Ll]iteral/
```

You can test relational expressions against built-up expressions. For example, the following pattern finds all records whose second field ($2) is at least 100 greater than the first field ($1):

```
$2 > $1 + 100
```

The following pattern finds records that contain an even number of fields:

```
NF % 2 == 0
```

Use the operators listed in Section 2.9 to build expressions.

You can use magnitude-comparison operators to test strings. For example, the following pattern finds records that begin with "s" or any character that appears after it to the end of the character set:

```
$0 >= "s"
```

You can combine two or more patterns by using the following Boolean operators:

&&	AND
||	Logical OR
!	NOT

For example, to prevent nonalphanumeric matches in the preceding example, you can combine two expressions as follows:

```
($0 >= "s" && $0 < "{")
```

(The left brace is the character immediately following the letter "z" in the ASCII

code.)

2.7 Using Pattern Ranges

You can use a pattern range to select a group of records to operate on. A pattern range consists of two patterns separated by a comma; the first pattern specifies the start of the range, and the second pattern specifies the end of the range. The `awk` program performs the associated action on all records in the range, including the records that match the two patterns. For example:

```
NR==100,NR==200 { print }
```

This program prints 101 records from the input file, beginning with record 100 and ending with record 200.

Using a pattern range does not disable other patterns from matching records within the range. However, because the input file is processed record by record, with each record being subject to all the actions appropriate to it before the next record is considered, the actions taken can appear to be out of sequence. For example:

```
2,4 { print }
/share/ { print "Found share" }
```

Apply this program to the following input file:

```
This is a test file
Line two
Try to share things
Line four
Last line of file
```

When this file is processed by `awk`, the output is as follows:

```
Line two
Try to share things
Found share
Line four
```

The second action is applied to record 3 before record 4 is examined to see if it matches the first pattern.

2.8 Actions in awk

An action can be a single command, such as `print`, or it can be a series of commands. An action can include tests to select records or parts of records; if desired, you can create a program that has no explicit patterns, relying instead on relational comparisons within its actions. Such a program can bear a strong resemblance to a C program; for example:

```
{
  if ($1 == 0) {
    print;
    printf("%5.2f\n", $2+$3)
  } else {
    printf("%5.2f\n", $1+$2)
```

```
  }
}
```

Note

The semicolon after the `print` command, which would be required in a C program, is not required by `awk`, but it does not cause an error.

2.9 Using Operators in an Action

Use the operators shown in Table 2-3 to build expressions within the action statement.

Table 2-3: Operators for awk Actions

Operator	Description	Example
+	Addition	2+3 = 5
+	Unary plus; placeholder	+4 = 4
-	Subtraction	7-3 = 4
-	Unary minus	-4 is negative 4
*	Multiplication	2*4 = 8
/	Division	6/3 = 2
%	Modulo (remaindering)	7%3 = 1
++	Increment	See the description following this table.
--	Decrement	See the description following this table.
+=	Increment by value	x+=y is equivalent to x = x+y
-=	Decrement by value	x-=y is equivalent to x = x-y
*=	Multiply by value	x*=y is equivalent to x = x*y
/=	Divide by value	x/=y is equivalent to x = x/y
%=	Modulo by value	x%=y is equivalent to x = x%y

The following example prints the sum of all the first fields and the sum of all the second fields in the input file:

```
{ s1 += $1; s2 += $2 }
END { print s1,s2 }
```

The position of the increment and decrement operators affects their interpretation. The expression `i++` evaluates the current contents of `i` and then increments `i`. The expression `++i` causes `awk` to increment `i` before evaluation. For example:

```
$ echo "3 3" | awk '{
```

```
>   print "$1 =", $1 "; $1++ =", $1++ "; new $1 =", $1
>   print "$2 =", $2 "; ++$2 =", ++$2 "; new $2 =", $2
> }'
$1 = 3; $1++ = 3; new $1 = 4
$2 = 3; ++$2 = 4; new $2 = 4
```

2.10 Using Functions Within an Action

The awk language includes the built-in functions listed in Table 2-4, Table 2-5, and Table 2-6; Additionally, awk lets you create additional functions as described after the tables.

Table 2-4: Built-In awk Mathematical Functions

Function	Description
atan2(*x*/*y*)	Returns the arctangent of the value specified by *x*/*y*.
cos(*expr*)	Returns the cosine of the value (in radians) specified by *expr*.
exp(*arg*)	Returns the natural antilogarithm (base *e*) of arg. For example, exp(0.693147) returns 2. See log(*arg*).
int(*arg*)	Returns the integer part of *arg*.
log(*arg*)	Returns the natural logarithm (base *e*) of *arg*. For example, log(2) returns 0.693147. See exp(*arg*).
rand	Returns a pseudorandom number (0 <= *n* < 1).
sin(*arg*)	Returns the sine of *arg*.
sqrt(*arg*)	Returns the square root of *arg*.
srand(*seed*)	Uses *seed* as the seed for a pseudorandom number sequence for subsequent calls to rand. If no seed is specified, the time of day is used. The return value is the previous seed.

Table 2-5: Built-In awk String Functions

Function	Description
gsub(*expr*,*s1*,*s2*)	Replaces every sequence of characters in string *s2* that matches the expression *expr* with the string specified by *s1*. If *s2* is not supplied, the current input record is used. Expression *expr* is reevaluated for each match. This function returns a value representing the number of replacements. See also sub(*expr*,*s1*,*s2*).

Function	Description
index(*s1*,*s2*)	Returns the character position in string *s1* where string *s2* occurs. If *s2* is not in *s1*, this function returns a zero.
length	Returns the length in characters of the current record.
length(*arg*)	Returns the length in characters of the string specified by *arg*. See length.
match(*s*,*expr*)	Returns the character position in string *s* where a match is found for the expression *expr*; sets the variable RSTART to the character position at which the match begins and RLENGTH to a value representing the length of the matched string. If no match is found, this function returns a zero.
split(*s*,*array*,*sep*)	Splits string *s* into consecutive elements of *array*[*1*]...[*n*] and returns the number of elements. The optional *sep* argument specifies a field separator other than the one currently in force (the contents of the FS variable).
sprintf(*f*,*e1*,*e2* ,...)	Returns (but does not print) a string containing the arguments *e1* and so on, formatted in the same manner as by the printf command.
strftime(*f*,*time*)	Formats *time* into a string as specified by *f*. The *time* value should be specified in the form returned by systime(). See the strftime(3) reference page for a list of the available format conversions.
sub(*expr*,*s1*,*s2*)	Replaces the first sequence of characters in string *s2* that matches the expression *expr* with the string specified by *s1*. If *s2* is not supplied, the current input record is used. This function returns a value representing the number of replacements (0 or 1). See also gsub(*expr*,*s1*,*s2*).
substr(*s*,*m*,*n*)	Returns the substring of *s* that begins at character position *m* and is *n* characters long. The first character in *s* is at position 1. If *n* is omitted or if the string is not long enough to supply *n* characters, the rest of the string is returned.
tolower(*s*)	Translates all uppercase letters in string *s* to lowercase. If there is no argument, the function operates on the current record.
toupper(*s*)	Translates all lowercase letters in string *s* to uppercase. If there is no argument, the function operates on the current record.

Table 2-6: Built-In Miscellaneous awk Functions

Function	Description
`close(`*arg*`)`	Closes the file or pipe named by `arg`.
`delete(`*array*`[`*sub*`])`	Deletes the element of *array* indicated by *sub*.
`system("`*command*`")`	Executes the system command specified and returns its exit status. The entire command must be enclosed in quotation marks to prevent `awk` from attempting to interpret it as one or more variable names.
`systime()`	Returns the current time of day as the number of seconds since midnight, January 1, 1970.

The `awk` language also lets you create functions by using the following syntax:

```
function name ( parameter-list ) {
   statements
}
```

The word `func` can be used in place of `function`. For functions that you create, the left parenthesis both in the function's definition and in its use must immediately follow the function name, with no intervening space. The names in the function declaration's parameter list are the formal parameters for use within the function. When you call a function, `awk` replaces these formal parameters with the values you supply in the calling statement. Functions can be recursive.

You can define local variables for a given function by declaring them as extra formal parameters; upon function entry, all local variables are initialized as empty strings or the number 0. To avoid visual confusion between real parameters and local variables, you can separate the local variables with extra spaces in the function declaration. For example:

```
function foo(in, out,      local1, local2) {
  local1 = "foo"
  local2 = "bar"

.
.
.
}
```

2.11 Using Control Structures in awk

The `awk` language provides the control structures listed in Table 2-7. Except where noted, these structures work exactly as they do in the C language. To perform several statements in a single control structure's action, enclose the statements in braces. If only a single statement is to be performed, the braces are optional. Each of the first two `if` structures in the following example includes a single statement to be executed; these structures are equivalent:

```
{
  if (x == y) print
  if (x == y) {
    print
```

```
  }
  if (x == y) {
    print $3
    printf("Sum = %d\n", x+z)
  }
}
```

Table 2-7: Control Structures in awk

Structure	Description
`if-else`	The condition in parentheses in an `if-else` structure is evaluated. If true, the statements following the `if` are performed. If false, the statements following the optional `else` keyword are performed.

`while`	The statements following the `while` statement are performed until the tested condition is true. The following example prints all the fields in the input records, one field per line:
	
	`{`
	`  i = 1`
	`  while(i<=NF) print $i++`
	`}`

`for`	The `for(`*expr1*`;`*expr2*`;`*expr3*`)` *statements* structure is equivalent to the following `while` construct:
	
	`{`
	`  `*expr1*
	`  while(`*expr2*`) {`
	`    `*statements*
	`    `*expr3*
	`  }`
	`}`
	
	The previous `while` example could also be written as follows:
	
	`{`
	`  for(i=1;i<=NF;++i) print $i`
	`}`

`break`	The `break` statement causes an immediate exit from an enclosing `while` or `for` loop.

Comments	Include comments in an `awk` program file to explain program logic. Comments begin with the number sign (`#`) and end with the end of the line. For example:
	
	`{`
	`  print x,y     # This is a comment`
	`}`

`continue`	The `continue` statement causes the next iteration of an enclosing loop to begin.
`getline`	The `getline` statement causes `awk` to discard the current input record, read the next input record, and continue scanning patterns from the present location.

Structure	Description
	By using `getline var`, you can assign the `getline` input to a variable; without *var*, the input is assigned to the current record.
`next`	The `next` statement causes `awk` to discard the current input record, read the next input record, and begin scanning patterns from the start of the program file.
`exit`	The `exit` statement causes the program to stop as if the end of the input occurred.

2.12 Performing Actions Before or After Processing the Input

The `awk` program recognizes two special pattern keywords that define the beginning (`BEGIN`) and the end (`END`) of the input file. `BEGIN` matches the beginning of the input before reading the first record. Therefore, `awk` performs any actions associated with this pattern once, before processing the input file. `BEGIN` must be the first pattern in the program file. For example, to change the field separator to a colon (`:`) for all records in the file, include the following line as the first line of the program file:

```
BEGIN { FS = ":" }
```

This example action works the same as using the `-F:` flag on the command line.

Similarly, `END` matches the end of the input file after processing the last record. Therefore, `awk` performs any actions associated with this pattern once, after processing the input file. `END` must be the last pattern in the program file. For example, to print the total number of records in the input file, include the following line as the last line in the program file:

```
END { print NR }
```

2.13 Concatenating Strings

You concatenate strings by placing their variable names together in an expression. For example, the command `print $1 $2` prints a string consisting of the first two fields from the current record, with no space between them. You can use variables, numeric operators, and functions when concatenating strings. (See Section 2.4.1 and Section 2.9 for information on variables and numeric operators.) The function `length($1 $2 $3)` returns the length in characters of the first three fields. (See Section 2.10 for a list of the functions in `awk`.) If the strings you want to concatenate are field variables (see Section 2.4.2), you are not required to separate the names with white space; the expression `$1$2` is identical to `$1 $2`.

2.14 Redirection and Pipes

Unless otherwise specified, `print` and `printf` statements write their output to the standard output file. You can redirect the output of any printing statement by using standard redirection operators. For example:

```
print $0, $3, amt >> "reportfile"
```

This example appends its output to a file named `reportfile` instead of writing to the standard output. (If `reportfile` does not exist before the first instance of redirection, it is created.) The output file name in this example is enclosed in quotation marks. The quotation marks are required to distinguish the file name from a variable name. You can mix writing to named files with writing to the standard output.

You can also pipe printed output through other commands. The following example pipes `awk`'s output through the `tr` command to convert all uppercase letters to lowercase letters:

```
print | "tr '[A-Z]' '[a-z]'"
```

As with redirection, the command to which you pipe the output must be enclosed in quotation marks. In `awk` you can redirect the input to `getline` using standard redirection operators, and you can supply the input to `getline` from a pipe. For example:

expr `| getline`

Here, *expr* is interpreted as a system command.

Only a limited number of files can be open for output. The `awk` program uses your default open file descriptor limit. For efficiency, however, you can use the `close(`*arg*`)` statement to close files that you have opened for output and no longer need.
