In this chapter:

- *Pattern Elements*
- *Using Shell Variables in Programs*
- *Actions*
- *Control Statements in Actions*
- *Built-in Variables*

6

Patterns, Actions, and Variables

As you have already seen, each *awk* statement consists of a pattern with an associated action. This chapter describes how you build patterns and actions, what kinds of things you can do within actions, and *awk*'s built-in variables.

The pattern-action rules and the statements available for use within actions form the core of *awk* programming. In a sense, everything covered in this text up to here has been the foundation that programs are built on top of. Now it's time to start building something useful.

Pattern Elements

Patterns in *awk* control the execution of rules—a rule is executed when its pattern matches the current input record. The following is a summary of the types of *awk* pattern types:

`/regular expression/`
: A regular expression. It matches when the text of the input record fits the regular expression. (See Chapter 2, *Regular Expressions*.)

`expression`
: A single expression. It matches when its value is nonzero (if a number) or non-null (if a string). (See the section "Expressions as Patterns" later in this chapter.)

`pat1``, `*`pat2`*
: A pair of patterns separated by a comma, specifying a range of records. The range includes both the initial record that matches *pat1* and the final record that matches *pat2*. (See the section "Specifying Record Ranges with Patterns" later in this chapter.)

`BEGIN`
`END`
: Special patterns for you to supply startup or cleanup actions for your *awk* program. (See the section "The BEGIN and END Special Patterns" later in this chapter.)

empty
: The empty pattern matches every input record. (See the section "The Empty Pattern" later in this chapter.)

Regular Expressions as Patterns

Regular expressions are one of the first kinds of patterns presented in this book. This kind of pattern is simply a regexp constant in the pattern part of a rule. Its meaning is `$0 ~ /`*`pattern`*`/`. The pattern matches when the input record matches the regexp. For example:

```
/foo|bar|baz/  { buzzwords++ }
END            { print buzzwords, "buzzwords seen" }
```

Expressions as Patterns

Any *awk* expression is valid as an *awk* pattern. The pattern matches if the expression's value is nonzero (if a number) or non-null (if a string). The expression is reevaluated each time the rule is tested against a new input record. If the expression uses fields such as `$1`, the value depends directly on the new input record's text; otherwise, it depends on only what has happened so far in the execution of the *awk* program.

Comparison expressions, using the comparison operators described in the section "Variable Typing and Comparison Expressions" in Chapter 5, *Expressions*, are a very common kind of pattern. Regexp matching and nonmatching are also very common expressions. The left operand of the `~` and `!~` operators is a string. The right operand is either a constant regular expression enclosed in slashes (`/`*`regexp`*`/`), or any expression whose string value is used as a dynamic regular expression (see the section "Using Dynamic Regexps" in Chapter 2). The following example prints the second field of each input record whose first field is precisely `foo`:

```
$ awk '$1 == "foo" { print $2 }' BBS-list
```

(There is no output, because there is no BBS site with the exact name `foo`.) Contrast this with the following regular expression match, which accepts any record with a first field that contains `foo`:

```
$ awk '$1 ~ /foo/ { print $2 }' BBS-list
555-1234
555-6699
555-6480
555-2127
```

A regexp constant as a pattern is also a special case of an expression pattern. The expression `/foo/` has the value one if `foo` appears in the current input record. Thus, as a pattern, `/foo/` matches any record containing `foo`.

Boolean expressions are also commonly used as patterns. Whether the pattern matches an input record depends on whether its subexpressions match. For example, the following command prints all the records in *BBS-list* that contain both `2400` and `foo`:

```
$ awk '/2400/ && /foo/' BBS-list
fooey       555-1234     2400/1200/300     B
```

The following command prints all records in *BBS-list* that contain *either* `2400` or `foo` (or both, of course):

```
$ awk '/2400/ || /foo/' BBS-list
alpo-net    555-3412     2400/1200/300     A
bites       555-1675     2400/1200/300     A
fooey       555-1234     2400/1200/300     B
foot        555-6699     1200/300          B
macfoo      555-6480     1200/300          A
sdace       555-3430     2400/1200/300     A
sabafoo     555-2127     1200/300          C
```

The following command prints all records in *BBS-list* that do *not* contain the string `foo`:

```
$ awk '! /foo/' BBS-list
aardvark    555-5553     1200/300          B
alpo-net    555-3412     2400/1200/300     A
barfly      555-7685     1200/300          A
bites       555-1675     2400/1200/300     A
camelot     555-0542     300               C
core        555-2912     1200/300          C
sdace       555-3430     2400/1200/300     A
```

The subexpressions of a Boolean operator in a pattern can be constant regular expressions, comparisons, or any other *awk* expressions. Range patterns are not expressions, so they cannot appear inside Boolean patterns. Likewise, the special patterns `BEGIN` and `END`, which never match any input record, are not expressions and cannot appear inside Boolean patterns.

Specifying Record Ranges with Patterns

A *range pattern* is made of two patterns separated by a comma, in the form *begpat*, *endpat*. It is used to match ranges of consecutive input records. The first pattern, *begpat*, controls where the range begins, while *endpat* controls where the pattern ends. For example, the following:

```
awk '$1 == "on", $1 == "off"' myfile
```

prints every record in *myfile* between `on`/`off` pairs, inclusive.

A range pattern starts out by matching *begpat* against every input record. When a record matches *begpat*, the range pattern is *turned on* and the range pattern matches this record as well. As long as the range pattern stays turned on, it automatically matches every input record read. The range pattern also matches *endpat* against every input record; when this succeeds, the range pattern is turned off again for the following record. Then the range pattern goes back to checking *begpat* against each record.

The record that turns on the range pattern and the one that turns it off both match the range pattern. If you don't want to operate on these records, you can write `if` statements in the rule's action to distinguish them from the records you are interested in.

It is possible for a pattern to be turned on and off by the same record. If the record satisfies both conditions, then the action is executed for just that record. For example, suppose there is text between two identical markers (e.g., the `%` symbol), each on its own line, that should be ignored. A first attempt would be to combine a range pattern that describes the delimited text with the `next` statement (not discussed yet, see the section "The next Statement" later in this chapter). This causes *awk* to skip any further processing of the current record and start over again with the next input record. Such a program looks like this:

```
/^%$/,/^%$/    { next }
               { print }
```

This program fails because the range pattern is both turned on and turned off by the first line, which just has a `%` on it. To accomplish this task, write the program in the following manner, using a flag:

```
/^%$/     { skip = ! skip; next }
skip == 1 { next } # skip lines with 'skip' set
```

In a range pattern, the comma (`,`) has the lowest precedence of all the operators (i.e., it is evaluated last). Thus, the following program attempts to combine a range pattern with another, simpler test:

```
echo Yes | awk '/1/,/2/ || /Yes/'
```

The intent of this program is `(/1/,/2/) || /Yes/`. However, *awk* interprets this as `/1/, (/2/ || /Yes/)`. This cannot be changed or worked around; range patterns do not combine with other patterns:

```
$ echo yes | gawk '(/1/,/2/) || /Yes/'
gawk: cmd. line:1: (/1/,/2/) || /Yes/
gawk: cmd. line:1:           ^ parse error
gawk: cmd. line:2: (/1/,/2/) || /Yes/
gawk: cmd. line:2:                   ^ unexpected newline
```

The BEGIN and END Special Patterns

All the patterns described so far are for matching input records. The `BEGIN` and `END` special patterns are different. They supply startup and cleanup actions for *awk* programs. `BEGIN` and `END` rules must have actions; there is no default action for these rules because there is no current record when they run. `BEGIN` and `END` rules are often referred to as "`BEGIN` and `END` blocks" by long-time *awk* programmers.

Startup and cleanup actions

A `BEGIN` rule is executed once only, before the first input record is read. Likewise, an `END` rule is executed once only, after all the input is read. For example:

```
$ awk '
> BEGIN { print "Analysis of \"foo\"" }
> /foo/ { ++n }
> END   { print "\"foo\" appears", n, "times." }' BBS-list
Analysis of "foo"
"foo" appears 4 times.
```

This program finds the number of records in the input file *BBS-list* that contain the string `foo`. The `BEGIN` rule prints a title for the report. There is no need to use the `BEGIN` rule to initialize the counter `n` to zero, since *awk* does this automatically (see the section "Variables" in Chapter 5). The second rule increments the variable `n` every time a record containing the pattern `foo` is read. The `END` rule prints the value of `n` at the end of the run.

The special patterns `BEGIN` and `END` cannot be used in ranges or with Boolean operators (indeed, they cannot be used with any operators). An *awk* program may have multiple `BEGIN` and/or `END` rules. They are executed in the order in which they appear: all the `BEGIN` rules at startup and all the `END` rules at termination. `BEGIN` and `END` rules may be intermixed with other rules. This feature was added in the 1987 version of *awk* and is included in the POSIX standard. The original (1978) version of *awk* required that the `BEGIN` rule was at the beginning of the program, and that the `END` rule was at the end, and only allowed one of each. This is no longer required, but it is a good idea to follow this template in terms of program organization and readability.

Multiple BEGIN and END rules are useful for writing library functions, because each library file can have its own BEGIN and/or END rule to do its own initialization and/or cleanup. The order in which library functions are named on the command-line controls the order in which their BEGIN and END rules are executed. Therefore, you have to be careful when writing such rules in library files so that the order in which they are executed doesn't matter. See the section "Command-Line Options" in Chapter 11, *Running awk and gawk*, for more information on using library functions. See Chapter 12, *A Library of awk Functions*, for a number of useful library functions.

If an *awk* program has only a BEGIN rule and no other rules, then the program exits after the BEGIN rule is run.* However, if an END rule exists, then the input is read, even if there are no other rules in the program. This is necessary in case the END rule checks the FNR and NR variables.

Input/Output from BEGIN and END rules

There are several (sometimes subtle) points to remember when doing I/O from a BEGIN or END rule. The first has to do with the value of $0 in a BEGIN rule. Because BEGIN rules are executed before any input is read, there simply is no input record, and therefore no fields, when executing BEGIN rules. References to $0 and the fields yield a null string or zero, depending upon the context. One way to give $0 a real value is to execute a getline command without a variable (see the section "Explicit Input with getline" in Chapter 3, *Reading Input Files*). Another way is simply to assign a value to $0.

The second point is similar to the first but from the other direction. Traditionally, due largely to implementation issues, $0 and NF were *undefined* inside an END rule. The POSIX standard specifies that NF is available in an END rule. It contains the number of fields from the last input record. Most probably due to an oversight, the standard does not say that $0 is also preserved, although logically one would think that it should be. In fact, *gawk* does preserve the value of $0 for use in END rules. Be aware, however, that Unix *awk*, and possibly other implementations, do not.

The third point follows from the first two. The meaning of print inside a BEGIN or END rule is the same as always: print $0. If $0 is the null string, then this prints an empty line. Many long time *awk* programmers use an unadorned print in BEGIN and END rules, to mean print "", relying on $0 being null. Although one might generally get away with this in BEGIN rules, it is a very bad idea in END rules, at least in *gawk*. It is also poor style, since if an empty line is needed in the output, the program should print one explicitly.

---

* The original version of *awk* used to keep reading and ignoring input until the end of the file was seen.

Finally, the `next` and `nextfile` statements are not allowed in a `BEGIN` rule, because the implicit read-a-record-and-match-against-the-rules loop has not started yet. Similarly, those statements are not valid in an `END` rule, since all the input has been read. (See the section "The next Statement" and section "Using gawk's nextfile Statement" later in this chapter.)

The Empty Pattern

An empty (i.e., nonexistent) pattern is considered to match *every* input record. For example, the program:

```
awk '{ print $1 }' BBS-list
```

prints the first field of every record.

Using Shell Variables in Programs

awk programs are often used as components in larger programs written in shell. For example, it is very common to use a shell variable to hold a pattern that the *awk* program searches for. There are two ways to get the value of the shell variable into the body of the *awk* program.

The most common method is to use shell quoting to substitute the variable's value into the program inside the script. For example, in the following program:

```
echo -n "Enter search pattern: "
read pattern
awk "/$pattern/ "'{ nmatches++ }
     END { print nmatches, "found" }' /path/to/data
```

the *awk* program consists of two pieces of quoted text that are concatenated together to form the program. The first part is double-quoted, which allows substitution of the `pattern` variable inside the quotes. The second part is single-quoted.

Variable substitution via quoting works, but can be potentially messy. It requires a good understanding of the shell's quoting rules (see the section "Shell-Quoting Issues" in Chapter 1, *Getting Started with awk*), and it's often difficult to correctly match up the quotes when reading the program.

A better method is to use *awk*'s variable assignment feature (see the section "Assigning Variables on the Command Line" in Chapter 5) to assign the shell variable's value to an *awk* variable's value. Then use dynamic regexps to match the pattern (see the section "Using Dynamic Regexps" in Chapter 2). The following shows how to redo the previous example using this technique:

```
echo -n "Enter search pattern: "
read pattern
awk -v pat="$pattern" '$0 ~ pat { nmatches++ }
                       END { print nmatches, "found" }' /path/to/data
```

Now, the *awk* program is just one single-quoted string. The assignment –v `pat="$pattern"` still requires double quotes, in case there is whitespace in the value of `$pattern`. The *awk* variable `pat` could be named `pattern` too, but that would be more confusing. Using a variable also provides more flexibility, since the variable can be used anywhere inside the program—for printing, as an array subscript, or for any other use—without requiring the quoting tricks at every point in the program.

Actions

An *awk* program or script consists of a series of rules and function definitions interspersed. (Functions are described later. See the section "User-Defined Functions" in Chapter 8, *Functions*.) A rule contains a pattern and an action, either of which (but not both) may be omitted. The purpose of the *action* is to tell *awk* what to do once a match for the pattern is found. Thus, in outline, an *awk* program generally looks like this:

```
[pattern] [{ action }]
[pattern] [{ action }]
...
function name(args) { ... }
...
```

An action consists of one or more *awk statements*, enclosed in curly braces (`{ }`). Each statement specifies one thing to do. The statements are separated by newlines or semicolons. The curly braces around an action must be used even if the action contains only one statement, or if it contains no statements at all. However, if you omit the action entirely, omit the curly braces as well. An omitted action is equivalent to `{ print $0 }`:

```
/foo/ { }   # match foo, do nothing -- empty action
/foo/       # match foo, print the record -- omitted action
```

The following types of statements are supported in *awk*:

Expressions
: Call functions or assign values to variables (see Chapter 5). Executing this kind of statement simply computes the value of the expression. This is useful when the expression has side effects (see the section "Assignment Expressions" in Chapter 5).

Control statements
: Specify the control flow of *awk* programs. The *awk* language gives you C-like constructs (`if`, `for`, `while`, and `do`) as well as a few special ones (see the section "Control Statements in Actions" later in this chapter).

Compound statements
: Consist of one or more statements enclosed in curly braces. A compound statement is used in order to put several statements together in the body of an `if`, `while`, `do`, or `for` statement.

Input statements
: Use the `getline` command (see the section "Explicit Input with getline" in Chapter 3). Also supplied in *awk* are the `next` statement (see the section "The next Statement" later in this chapter) and the `nextfile` statement (see the section "Using gawk's nextfile Statement" later in this chapter).

Output statements
: Such as `print` and `printf`. See Chapter 4, *Printing Output*.

Deletion statements
: For deleting array elements. See the section "The delete Statement" in Chapter 7, *Arrays in awk*.

Control Statements in Actions

Control statements, such as `if`, `while`, and so on, control the flow of execution in *awk* programs. Most of the control statements in *awk* are patterned on similar statements in C.

All the control statements start with special keywords, such as `if` and `while`, to distinguish them from simple expressions. Many control statements contain other statements. For example, the `if` statement contains another statement that may or may not be executed. The contained statement is called the *body*. To include more than one statement in the body, group them into a single *compound statement* with curly braces, separating them with newlines or semicolons.

The if-else Statement

The `if-else` statement is *awk*'s decision-making statement. It looks like this:

```
if (condition) then-body [else else-body]
```

The *condition* is an expression that controls what the rest of the statement does. If the *condition* is true, *then-body* is executed; otherwise, *else-body* is executed. The `else` part of the statement is optional. The condition is considered false if its value is zero or the null string; otherwise, the condition is true. Refer to the following:

```
if (x % 2 == 0)
    print "x is even"
else
    print "x is odd"
```

In this example, if the expression `x % 2 == 0` is true (that is, if the value of `x` is evenly divisible by two), then the first `print` statement is executed; otherwise, the second `print` statement is executed. If the `else` keyword appears on the same line as *then-body* and *then-body* is not a compound statement (i.e., not surrounded by curly braces), then a semicolon must separate *then-body* from the `else`. To illustrate this, the previous example can be rewritten as:

```
if (x % 2 == 0) print "x is even"; else
    print "x is odd"
```

If the `;` is left out, *awk* can't interpret the statement and it produces a syntax error. Don't actually write programs this way, because a human reader might fail to see the `else` if it is not the first thing on its line.

The while Statement

In programming, a *loop* is a part of a program that can be executed two or more times in succession. The `while` statement is the simplest looping statement in *awk*. It repeatedly executes a statement as long as a condition is true. For example:

```
while (condition)
    body
```

body is a statement called the *body* of the loop, and *condition* is an expression that controls how long the loop keeps running. The first thing the `while` statement does is test the *condition*. If the *condition* is true, it executes the statement *body*. After *body* has been executed, *condition* is tested again, and if it is still true, *body* is executed again. This process repeats until the *condition* is no longer true. If the *condition* is initially false, the body of the loop is never executed and *awk* continues with the statement following the loop. This example prints the first three fields of each record, one per line:

```
awk '{ i = 1
       while (i <= 3) {
           print $i
           i++
       }
}' inventory-shipped
```

The body of this loop is a compound statement enclosed in braces, containing two statements. The loop works in the following manner: first, the value of `i` is set to one. Then, the `while` statement tests whether `i` is less than or equal to three. This is true when `i` equals one, so the `i`-th field is printed. Then the `i++` increments the value of `i` and the loop repeats. The loop terminates when `i` reaches four.

A newline is not required between the condition and the body; however using one makes the program clearer unless the body is a compound statement or else is very simple. The newline after the open-brace that begins the compound statement is not required either, but the program is harder to read without it.

The do-while Statement

The `do` loop is a variation of the `while` looping statement. The `do` loop executes the *body* once and then repeats the *body* as long as the *condition* is true. It looks like this:

```
do
    body
while (condition)
```

Even if the *condition* is false at the start, the *body* is executed at least once (and only once, unless executing *body* makes *condition* true). Contrast this with the corresponding `while` statement:

```
while (condition)
    body
```

This statement does not execute *body* even once if the *condition* is false to begin with. The following is an example of a `do` statement:

```
{    i = 1
     do {
         print $0
         i++
     } while (i <= 10)
}
```

This program prints each input record 10 times. However, it isn't a very realistic example, since in this case an ordinary `while` would do just as well. This situation reflects actual experience; only occasionally is there a real use for a `do` statement.

The for Statement

The `for` statement makes it more convenient to count iterations of a loop. The general form of the `for` statement looks like this:

```
for (initialization; condition; increment)
    body
```

The *initialization*, *condition*, and *increment* parts are arbitrary *awk* expressions, and *body* stands for any *awk* statement.

The `for` statement starts by executing *initialization*. Then, as long as the *condition* is true, it repeatedly executes *body* and then *increment*. Typically, *initialization* sets a variable to either zero or one, *increment* adds one to it, and *condition* compares it against the desired number of iterations. For example:

```
awk '{ for (i = 1; i <= 3; i++)
          print $i
}' inventory-shipped
```

This prints the first three fields of each input record, with one field per line.

It isn't possible to set more than one variable in the *initialization* part without using a multiple assignment statement such as `x = y = 0`. This makes sense only if all the initial values are equal. (But it is possible to initialize additional variables by writing their assignments as separate statements preceding the `for` loop.)

The same is true of the *increment* part. Incrementing additional variables requires separate statements at the end of the loop. The C compound expression, using C's comma operator, is useful in this context but it is not supported in *awk*.

Most often, *increment* is an increment expression, as in the previous example. But this is not required; it can be any expression whatsoever. For example, the following statement prints all the powers of two between 1 and 100:

```
for (i = 1; i <= 100; i *= 2)
    print i
```

If there is nothing to be done, any of the three expressions in the parentheses following the `for` keyword may be omitted. Thus, `for (; x > 0;)` is equivalent to `while (x > 0)`. If the *condition* is omitted, it is treated as true, effectively yielding an *infinite loop* (i.e., a loop that never terminates).

In most cases, a `for` loop is an abbreviation for a `while` loop, as shown here:

```
initialization
while (condition) {
    body
    increment
}
```

The only exception is when the `continue` statement (see the section "The continue Statement" later in this chapter) is used inside the loop. Changing a `for` statement to a `while` statement in this way can change the effect of the `continue` statement inside the loop.

The *awk* language has a `for` statement in addition to a `while` statement because a `for` loop is often both less work to type and more natural to think of. Counting the number of iterations is very common in loops. It can be easier to think of this counting as part of looping rather than as something to do inside the loop.

The break Statement

The `break` statement jumps out of the innermost `for`, `while`, or `do` loop that encloses it. The following example finds the smallest divisor of any integer, and also identifies prime numbers:

```
# find smallest divisor of num
{
    num = $1
    for (div = 2; div*div <= num; div++)
        if (num % div == 0)
            break
    if (num % div == 0)
        printf "Smallest divisor of %d is %d\n", num, div
    else
        printf "%d is prime\n", num
}
```

When the remainder is zero in the first `if` statement, *awk* immediately *breaks out* of the containing `for` loop. This means that *awk* proceeds immediately to the statement following the loop and continues processing. (This is very different from the `exit` statement, which stops the entire *awk* program. See the section "The exit Statement" later in this chapter.)

Th following program illustrates how the *condition* of a `for` or `while` statement could be replaced with a `break` inside an `if`:

```
# find smallest divisor of num
{
    num = $1
    for (div = 2; ; div++) {
        if (num % div == 0) {
            printf "Smallest divisor of %d is %d\n", num, div
            break
        }
        if (div*div > num) {
            printf "%d is prime\n", num
            break
        }
    }
}
```

The `break` statement has no meaning when used outside the body of a loop. However, although it was never documented, historical implementations of *awk* treated the `break` statement outside of a loop as if it were a `next` statement (see the section "The next Statement" later in this chapter). Recent versions of Unix *awk* no longer allow this usage. *gawk* supports this use of `break` only if *––traditional* has been specified on the command line (see the section "Command-Line Options" in Chapter 11). Otherwise, it is treated as an error, since the POSIX standard specifies that `break` should only be used inside the body of a loop. (d.c.)

The continue Statement

As with `break`, the `continue` statement is used only inside `for`, `while`, and `do` loops. It skips over the rest of the loop body, causing the next cycle around the loop to begin immediately. Contrast this with `break`, which jumps out of the loop altogether.

The `continue` statement in a `for` loop directs *awk* to skip the rest of the body of the loop and resume execution with the increment-expression of the `for` statement. The following program illustrates this fact:

```
BEGIN {
    for (x = 0; x <= 20; x++) {
        if (x == 5)
            continue
        printf "%d ", x
    }
    print ""
}
```

This program prints all the numbers from 0 to 20—except for 5, for which the `printf` is skipped. Because the increment `x++` is not skipped, `x` does not remain stuck at 5. Contrast the `for` loop from the previous example with the following `while` loop:

```
BEGIN {
    x = 0
    while (x <= 20) {
        if (x == 5)
            continue
        printf "%d ", x
        x++
    }
    print ""
}
```

This program loops forever once `x` reaches 5.

The `continue` statement has no meaning when used outside the body of a loop. Historical versions of *awk* treated a `continue` statement outside a loop the same way they treated a `break` statement outside a loop: as if it were a `next` statement. Recent versions of Unix *awk* no longer work this way, and *gawk* allows it only if *––traditional* is specified on the command line (see the section "Command-Line Options" in Chapter 11). Just like the `break` statement, the POSIX standard specifies that `continue` should only be used inside the body of a loop. (d.c.)

The next Statement

The `next` statement forces *awk* to immediately stop processing the current record and go on to the next record. This means that no further rules are executed for the current record, and the rest of the current rule's action isn't executed.

Contrast this with the effect of the `getline` function (see the section "Explicit Input with getline" in Chapter 3). That also causes *awk* to read the next record immediately, but it does not alter the flow of control in any way (i.e., the rest of the current action executes with a new input record).

At the highest level, *awk* program execution is a loop that reads an input record and then tests each rule's pattern against it. If you think of this loop as a `for` statement whose body contains the rules, then the `next` statement is analogous to a `continue` statement. It skips to the end of the body of this implicit loop and executes the increment (which reads another record).

For example, suppose an *awk* program works only on records with four fields, and it shouldn't fail when given bad input. To avoid complicating the rest of the program, write a "weed out" rule near the beginning, in the following manner:

```
NF != 4 {
    err = sprintf("%s:%d: skipped: NF != 4\n", FILENAME, FNR)
    print err > "/dev/stderr"
    next
}
```

Because of the `next` statement, the program's subsequent rules won't see the bad record. The error message is redirected to the standard error output stream, as error messages should be. For more detail see the section "Special Filenames in gawk" in Chapter 4.

According to the POSIX standard, the behavior is undefined if the `next` statement is used in a `BEGIN` or `END` rule. *gawk* treats it as a syntax error. Although POSIX permits it, some other *awk* implementations don't allow the `next` statement inside function bodies (see the section "User-Defined Functions" in Chapter 8). Just as with any other `next` statement, a `next` statement inside a function body reads the next record and starts processing it with the first rule in the program. If the `next` statement causes the end of the input to be reached, then the code in any `END` rules is executed. See the section "The BEGIN and END Special Patterns" earlier in this chapter.

Using gawk's nextfile Statement

gawk provides the `nextfile` statement, which is similar to the `next` statement. However, instead of abandoning processing of the current record, the `nextfile` statement instructs *gawk* to stop processing the current datafile.

The `nextfile` statement is a *gawk* extension. In most other *awk* implementations, or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), `nextfile` is not special.

Upon execution of the `nextfile` statement, `FILENAME` is updated to the name of the next datafile listed on the command line, `FNR` is reset to one, `ARGIND` is incremented, and processing starts over with the first rule in the program. (`ARGIND` hasn't been introduced yet. See the section "Built-in Variables" later in this chapter.) If the `nextfile` statement causes the end of the input to be reached, then the code in any `END` rules is executed. See the section "The BEGIN and END Special Patterns" earlier in this chapter.

The `nextfile` statement is useful when there are many datafiles to process but it isn't necessary to process every record in every file. Normally, in order to move on to the next datafile, a program has to continue scanning the unwanted records. The `nextfile` statement accomplishes this much more efficiently.

While one might think that `close(FILENAME)` would accomplish the same as `nextfile`, this isn't true. `close` is reserved for closing files, pipes, and coprocesses that are opened with redirections. It is not related to the main processing that *awk* does with the files listed in `ARGV`.

If it's necessary to use an *awk* version that doesn't support `nextfile`, see the section "Implementing nextfile as a Function" in Chapter 12 for a user-defined function that simulates the `nextfile` statement.

The current version of the Bell Laboratories *awk* (see the section "Other Freely Available awk Implementations" in Appendix B, *Installing gawk*) also supports `nextfile`. However, it doesn't allow the `nextfile` statement inside function bodies (see the section "User-Defined Functions" in Chapter 8). *gawk* does; a `nextfile` inside a function body reads the next record and starts processing it with the first rule in the program, just as any other `nextfile` statement.

Versions of *gawk* prior to 3.0 used two words (`next file`) for the `nextfile` statement. In Version 3.0, this was changed to one word, because the treatment of `file` was inconsistent. When it appeared after `next`, `file` was a keyword; otherwise, it was a regular identifier. The old usage is no longer accepted; `next file` generates a syntax error.

The exit Statement

The `exit` statement causes *awk* to immediately stop executing the current rule and to stop processing input; any remaining input is ignored. The `exit` statement is written as follows:

```
exit [return code]
```

When an `exit` statement is executed from a `BEGIN` rule, the program stops processing everything immediately. No input records are read. However, if an `END` rule is present, as part of executing the `exit` statement, the `END` rule is executed (see the section "The BEGIN and END Special Patterns" earlier in this chapter). If `exit` is used as part of an `END` rule, it causes the program to stop immediately.

An `exit` statement that is not part of a `BEGIN` or `END` rule stops the execution of any further automatic rules for the current record, skips reading any remaining input records, and executes the `END` rule if there is one.

In such a case, if you don't want the `END` rule to do its job, set a variable to nonzero before the `exit` statement and check that variable in the `END` rule. See the section "Assertions" in Chapter 12 for an example that does this.

If an argument is supplied to `exit`, its value is used as the exit status code for the *awk* process. If no argument is supplied, `exit` returns status zero (success). In the case where an argument is supplied to a first `exit` statement, and then `exit` is called a second time from an `END` rule with no argument, *awk* uses the previously supplied exit value. (d.c.)

For example, suppose an error condition occurs that is difficult or impossible to handle. Conventionally, programs report this by exiting with a nonzero status. An *awk* program can do this using an `exit` statement with a nonzero argument, as shown in the following example:

```
BEGIN {
    if (("date" | getline date_now) <= 0) {
        print "Can't get system date" > "/dev/stderr"
        exit 1
    }
    print "current date is", date_now
    close("date")
}
```

Built-in Variables

Most *awk* variables are available to use for your own purposes; they never change unless your program assigns values to them, and they never affect anything unless your program examines them. However, a few variables in *awk* have special built-in meanings. *awk* examines some of these automatically, so that they enable you to tell *awk* how to do certain things. Others are set automatically by *awk*, so that they carry information from the internal workings of *awk* to your program.

This section documents all the built-in variables of *gawk*, most of which are also documented in the chapters describing their areas of activity.

Built-in Variables That Control awk

The following is an alphabetical list of variables that you can change to control how *awk* does certain things. The variables that are specific to *gawk* are marked with a pound sign (`#`):

`BINMODE #`

On non-POSIX systems, this variable specifies use of binary mode for all I/O. Numeric values of one, two, or three specify that input files, output files, or all files, respectively, should use binary I/O. Alternatively, string values of `"r"` or `"w"` specify that input files and output files, respectively, should use binary I/O. A string value of `"rw"` or `"wr"` indicates that all files should use binary I/O. Any other string value is equivalent to `"rw"`, but *gawk* generates a warning message. `BINMODE` is described in more detail in the section "Using gawk on PC Operating Systems" in Appendix B.

This variable is a *gawk* extension. In other *awk* implementations (except *mawk*, see the section "Other Freely Available awk Implementations" in Appendix B), or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), it is not special.

`CONVFMT`

This string controls conversion of numbers to strings (see the section "Conversion of Strings and Numbers" in Chapter 5). It works by being passed, in effect, as the first argument to the `sprintf` function (see the section "String-Manipulation Functions" in Chapter 8). Its default value is `"%.6g"`. `CONVFMT` was introduced by the POSIX standard.

`FIELDWIDTHS #`

This is a space-separated list of columns that tells *gawk* how to split input with fixed columnar boundaries. Assigning a value to `FIELDWIDTHS` overrides the use of `FS` for field splitting. See the section "Reading Fixed-Width Data" in Chapter 3 for more information.

If *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), then `FIELDWIDTHS` has no special meaning, and field-splitting operations occur based exclusively on the value of `FS`.

`FS`

This is the input field separator (see the section "Specifying How Fields Are Separated" in Chapter 3). The value is a single-character string or a multi-character regular expression that matches the separations between fields in an input record. If the value is the null string (`""`), then each character in the record becomes a separate field. (This behavior is a *gawk* extension. POSIX *awk* does not specify the behavior when `FS` is the null string.)

The default value is `" "`, a string consisting of a single space. As a special exception, this value means that any sequence of spaces, tabs, and/or newlines is a single separator.* It also causes spaces, tabs, and newlines at the beginning and end of a record to be ignored.

You can set the value of `FS` on the command line using the *–F* option:

```
awk -F, 'program' input-files
```

If *gawk* is using `FIELDWIDTHS` for field splitting, assigning a value to `FS` causes *gawk* to return to the normal, `FS`-based field splitting. An easy way to do this is to simply say `FS = FS`, perhaps with an explanatory comment.

`IGNORECASE #`

If `IGNORECASE` is nonzero or non-null, then all string comparisons and all regular expression matching are case independent. Thus, regexp matching with `~` and `!~`, as well as the `gensub`, `gsub`, `index`, `match`, `split`, and `sub` functions, record termination with `RS`, and field splitting with `FS`, all ignore case when doing their particular regexp operations. However, the value of `IGNORECASE` does *not* affect array subscripting. See the section "Case Sensitivity in Matching" in Chapter 2.

If *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), then `IGNORECASE` has no special meaning. Thus, string and regexp operations are always case-sensitive.

`LINT #`

When this variable is true (nonzero or non-null), *gawk* behaves as if the *––lint* command-line option is in effect. (see the section "Command-Line Options" in Chapter 11). With a value of `"fatal"`, lint warnings become fatal errors. Any other true value prints nonfatal warnings. Assigning a false value to `LINT` turns off the lint warnings.

---

* In POSIX *awk*, newline does not count as whitespace.

This variable is a *gawk* extension. It is not special in other *awk* implementations. Unlike the other special variables, changing `LINT` does affect the production of lint warnings, even if *gawk* is in compatibility mode. Much as the *--lint* and *--traditional* options independently control different aspects of *gawk*'s behavior, the control of lint warnings during program execution is independent of the flavor of *awk* being executed.

`OFMT`
: This string controls conversion of numbers to strings (see the section "Conversion of Strings and Numbers" in Chapter 5) for printing with the `print` statement. It works by being passed as the first argument to the `sprintf` function (see the section "String-Manipulation Functions" in Chapter 8). Its default value is `"%.6g"`. Earlier versions of *awk* also used `OFMT` to specify the format for converting numbers to strings in general expressions; this is now done by `CONVFMT`.

`OFS`
: This is the output field separator (see the section "Output Separators" in Chapter 4). It is output between the fields printed by a `print` statement. Its default value is `" "`, a string consisting of a single space.

`ORS`
: This is the output record separator. It is output at the end of every `print` statement. Its default value is `"\n"`, the newline character. (See the section "Output Separators" in Chapter 4.)

`RS`
: This is *awk*'s input record separator. Its default value is a string containing a single newline character, which means that an input record consists of a single line of text. It can also be the null string, in which case records are separated by runs of blank lines. If it is a regexp, records are separated by matches of the regexp in the input text. (See the section "How Input Is Split into Records" in Chapter 3.)

  The ability for `RS` to be a regular expression is a *gawk* extension. In most other *awk* implementations, or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), just the first character of `RS`'s value is used.

`SUBSEP`
: This is the subscript separator. It has the default value of `"\034"` and is used to separate the parts of the indices of a multidimensional array. Thus, the expression `foo["A", "B"]` really accesses `foo["A\034B"]` (see the section "Multidimensional Arrays" in Chapter 7).

`TEXTDOMAIN` #
: This variable is used for internationalization of programs at the *awk* level. It sets the default text domain for specially marked string constants in the source text, as well as for the `dcgettext` and `bindtextdomain` functions (see Chapter 9, *Internationalization with gawk*). The default value of `TEXTDOMAIN` is `"messages"`.

  This variable is a *gawk* extension. In other *awk* implementations, or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), it is not special.

Built-in Variables That Convey Information

The following is an alphabetical list of variables that *awk* sets automatically on certain occasions in order to provide information to your program. The variables that are specific to *gawk* are marked with a pound sign (`#`):

`ARGC`, `ARGV`
: The command-line arguments available to *awk* programs are stored in an array called `ARGV`. `ARGC` is the number of command-line arguments present. See the section "Other Command-Line Arguments" in Chapter 11. Unlike most *awk* arrays, `ARGV` is indexed from 0 to `ARGC` − 1. In the following example:

```
$ awk 'BEGIN {
>         for (i = 0; i < ARGC; i++)
>             print ARGV[i]
>      }' inventory-shipped BBS-list
awk
inventory-shipped
BBS-list
```

  `ARGV[0]` contains `"awk"`, `ARGV[1]` contains `"inventory-shipped"`, and `ARGV[2]` contains `"BBS-list"`. The value of `ARGC` is three, one more than the index of the last element in `ARGV`, because the elements are numbered from zero.

  The names `ARGC` and `ARGV`, as well as the convention of indexing the array from 0 to `ARGC` − 1, are derived from the C language's method of accessing command-line arguments.

  The value of `ARGV[0]` can vary from system to system. Also, you should note that the program text is *not* included in `ARGV`, nor are any of *awk*'s command-line options. See the section "Using ARGC and ARGV" later in this chapter for information about how *awk* uses these variables.

`ARGIND` #
: The index in `ARGV` of the current file being processed. Every time *gawk* opens a new datafile for processing, it sets `ARGIND` to the index in `ARGV` of the filename. When *gawk* is processing the input files, `FILENAME == ARGV[ARGIND]` is always true.

This variable is useful in file processing; it allows you to tell how far along you are in the list of datafiles as well as to distinguish between successive instances of the same filename on the command line.

While you can change the value of `ARGIND` within your *awk* program, *gawk* automatically sets it to a new value when the next file is opened.

This variable is a *gawk* extension. In other *awk* implementations, or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), it is not special.

`ENVIRON`

An associative array that contains the values of the environment. The array indices are the environment variable names; the elements are the values of the particular environment variables. For example, `ENVIRON["HOME"]` might be */home/arnold*. Changing this array does not affect the environment passed on to any programs that *awk* may spawn via redirection or the `system` function.

Some operating systems may not have environment variables. On such systems, the `ENVIRON` array is empty (except for `ENVIRON["AWKPATH"]`; see the section "The AWKPATH Environment Variable" in Chapter 11).

`ERRNO` #

If a system error occurs during a redirection for `getline`, during a read for `getline`, or during a `close` operation, then `ERRNO` contains a string describing the error.

This variable is a *gawk* extension. In other *awk* implementations, or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), it is not special.

`FILENAME`

The name of the file that *awk* is currently reading. When no datafiles are listed on the command line, *awk* reads from the standard input and `FILENAME` is set to `"-"`. `FILENAME` is changed each time a new file is read (see Chapter 3). Inside a `BEGIN` rule, the value of `FILENAME` is `""`, since there are no input files being processed yet.* (d.c.) Note, though, that using `getline` (see the section "Explicit Input with getline" in Chapter 3) inside a `BEGIN` rule can give `FILENAME` a value.

`FNR`

The current record number in the current file. `FNR` is incremented each time a new record is read (see the section "Explicit Input with getline" in Chapter 3). It is reinitialized to zero each time a new input file is started.

---

* Some early implementations of Unix *awk* initialized `FILENAME` to `"-"`, even if there were datafiles to be processed. This behavior was incorrect and should not be relied upon in your programs.

`NF`
: The number of fields in the current input record. `NF` is set each time a new record is read, when a new field is created or when `$0` changes (see the section "Examining Fields" in Chapter 3).

`NR`
: The number of input records *awk* has processed since the beginning of the program's execution (see the section "How Input Is Split into Records" in Chapter 3). `NR` is incremented each time a new record is read.

`PROCINFO #`
: The elements of this array provide access to information about the running *awk* program. The following elements (listed alphabetically) are guaranteed to be available:

  `PROCINFO["egid"]`
  : The value of the `getegid` system call.

  `PROCINFO["euid"]`
  : The value of the `geteuid` system call.

  `PROCINFO["FS"]`
  : This is `"FS"` if field splitting with `FS` is in effect, or it is `"FIELDWIDTHS"` if field splitting with `FIELDWIDTHS` is in effect.

  `PROCINFO["gid"]`
  : The value of the `getgid` system call.

  `PROCINFO["pgrpid"]`
  : The process group ID of the current process.

  `PROCINFO["pid"]`
  : The process ID of the current process.

  `PROCINFO["ppid"]`
  : The parent process ID of the current process.

  `PROCINFO["uid"]`
  : The value of the `getuid` system call.

  On some systems, there may be elements in the array, `"group1"` through `"group`*N*`"` for some *N*. *N* is the number of supplementary groups that the process has. Use the `in` operator to test for these elements (see the section "Referring to an Array Element" in Chapter 7).

  This array is a *gawk* extension. In other *awk* implementations, or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), it is not special.

`RLENGTH`
: The length of the substring matched by the `match` function (see the section "String-Manipulation Functions" in Chapter 8). `RLENGTH` is set by invoking the `match` function. Its value is the length of the matched string, or −1 if no match is found.

`RSTART`
: The start index in characters of the substring that is matched by the `match` function (see the section "String-Manipulation Functions" in Chapter 8). `RSTART` is set by invoking the `match` function. Its value is the position of the string where the matched substring starts, or zero if no match was found.

`RT #`
: This is set each time a record is read. It contains the input text that matched the text denoted by `RS`, the record separator.

  This variable is a *gawk* extension. In other *awk* implementations, or if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11), it is not special.

Changing NR and FNR

awk increments `NR` and `FNR` each time it reads a record, instead of setting them to the absolute value of the number of records read. This means that a program can change these variables and their new values are incremented for each record. (d.c.) This is demonstrated in the following example:

```
$ echo '1
> 2
> 3
> 4' | awk 'NR == 2 { NR = 17 }
> { print NR }'
1
17
18
19
```

Before `FNR` was added to the *awk* language (see the section "Major Changes Between V7 and SVR3.1" in Appendix A, *The Evolution of the awk Language*), many *awk* programs used this feature to track the number of records in a file by resetting `NR` to zero when `FILENAME` changed.

Using ARGC and ARGV

The previous section "Built-in Variables That Convey Information" presented the following program describing the information contained in ARGC and ARGV:

```
$ awk 'BEGIN {
>         for (i = 0; i < ARGC; i++)
>             print ARGV[i]
>       }' inventory-shipped BBS-list
awk
inventory-shipped
BBS-list
```

In this example, ARGV[0] contains awk, ARGV[1] contains inventory-shipped, and ARGV[2] contains BBS-list. Notice that the *awk* program is not entered in ARGV. The other special command-line options, with their arguments, are also not entered. This includes variable assignments done with the *–v* option (see the section "Command-Line Options" in Chapter 11). Normal variable assignments on the command line *are* treated as arguments and do show up in the ARGV array:

```
$ cat showargs.awk
BEGIN {
    printf "A=%d, B=%d\n", A, B
    for (i = 0; i < ARGC; i++)
        printf "\tARGV[%d] = %s\n", i, ARGV[i]
}
END   { printf "A=%d, B=%d\n", A, B }
$ awk -v A=1 -f showargs.awk B=2 /dev/null
A=1, B=0
        ARGV[0] = awk
        ARGV[1] = B=2
        ARGV[2] = /dev/null
A=1, B=2
```

A program can alter ARGC and the elements of ARGV. Each time *awk* reaches the end of an input file, it uses the next element of ARGV as the name of the next input file. By storing a different string there, a program can change which files are read. Use – to represent the standard input. Storing additional elements and incrementing ARGC causes additional files to be read.

If the value of ARGC is decreased, that eliminates input files from the end of the list. By recording the old value of ARGC elsewhere, a program can treat the eliminated arguments as something other than filenames.

To eliminate a file from the middle of the list, store the null string ("") into ARGV in place of the file's name. As a special feature, *awk* ignores filenames that have been replaced with the null string. Another option is to use the delete statement to remove elements from ARGV (see the section "The delete Statement" in Chapter 7).

All of these actions are typically done in the `BEGIN` rule, before actual processing of the input begins. See the section "Splitting a Large File into Pieces" and the section "Duplicating Output into Multiple Files" in Chapter 13, *Practical awk Programs*, for examples of each way of removing elements from `ARGV`. The following fragment processes `ARGV` in order to examine, and then remove, command-line options:

```
BEGIN {
    for (i = 1; i < ARGC; i++) {
        if (ARGV[i] == "-v")
            verbose = 1
        else if (ARGV[i] == "-d")
            debug = 1
        else if (ARGV[i] ~ /^-?/) {
            e = sprintf("%s: unrecognized option -- %c",
                    ARGV[0], substr(ARGV[i], 1, ,1))
            print e > "/dev/stderr"
        } else
            break
        delete ARGV[i]
    }
}
```

To actually get the options into the *awk* program, end the *awk* options with `--` and then supply the *awk* program's options, in the following manner:

```
awk -f myprog -- -v -d file1 file2 ...
```

This is not necessary in *gawk*. Unless `--posix` has been specified, *gawk* silently puts any unrecognized options into `ARGV` for the *awk* program to deal with. As soon as it sees an unknown option, *gawk* stops looking for other options that it might otherwise recognize. The previous example with *gawk* would be:

```
gawk -f myprog -d -v file1 file2 ...
```

Because *-d* is not a valid *gawk* option, it and the following *-v* are passed on to the *awk* program.