CHAPTER 9
Enough awk to Be Dangerous

The awk programming language was designed to simplify many common text processing tasks. In this chapter, we present a subset that suffices for most of the shell scripts that we use in this book.

For an extended treatment of the awk language, consult any of the books on awk listed in the Bibliography. If GNU gawk is installed on your system, then its manual should be available in the online info system.*

All Unix systems have at least one awk implementation. When the language was significantly extended in the mid-1980s, some vendors kept the old implementation as awk, and sometimes also as oawk, and then named the new one nawk. IBM AIX and Sun Solaris both continue that practice, but most others now provide only the new one. Solaris has a POSIX-compliant version in /usr/xpg4/bin/awk. In this book, we consider only the extended language and refer to it as awk, even though you might have to use nawk, gawk, or mawk on your system.

We must confess here to a strong bias about awk. We like it. A lot. We have implemented, maintained, ported, written about, and used the language for many years. Even though many awk programs are short, some of our larger awk programs are thousands of lines long. The simplicity and power of awk often make it just the right tool for the job, and we seldom encounter a text processing task in which we need a feature that is not already in the language, or cannot be readily implemented. When we have on occasion rewritten an awk program in a conventional programming language like C or C++, the result was usually much longer, and much harder to debug, even if it did run somewhat faster.

Unlike most other scripting languages, awk enjoys multiple implementations, a healthy situation that encourages adherence to a common language base and that

* The GNU documentation reader, info, is part of the *texinfo* package available at *ftp://ftp.gnu.org/gnu/texinfo/*. The emacs text editor also can be used to access the same documentation: type Ctrl-H i in an emacs session to get started.

permits users to switch freely from one to another. Also, unlike other scripting languages, awk is part of POSIX, and there are implementations for non-Unix operating systems.

If your local version of awk is substandard, get one of the free implementations listed in Table 9-1. All of these programs are very portable and easy to install. gawk has served as a testbed for several interesting new built-in functions and language features, including network I/O, and also for profiling, internationalization, and portability checking.

Table 9-1. Freely available awk versions

Program	Location
Bell Labs awk	*http://cm.bell-labs.com/who/bwk/awk.tar.gz*
gawk	*ftp://ftp.gnu.org/gnu/gawk/*
mawk	*ftp://ftp.whidbey.net/pub/brennan/mawk-1.3.3.tar.gz*
awka	*http://awka.sourceforge.net/* (awk-to-C translator)

9.1 The awk Command Line

An awk invocation can define variables, supply the program, and name the input files:

```
awk [ -F fs ] [ -v var=value ... ] 'program' [ -- ] \
      [ var=value ... ] [ file(s) ]

awk [ -F fs ] [ -v var=value ... ] -f programfile [ -- ] \
      [ var=value ... ] [ file(s) ]
```

Short programs are usually provided directly on the command line, whereas longer ones are relegated to files selected by the -f option. That option may be repeated, in which case the complete program is the concatenation of the specified program files. This is a convenient way to include libraries of shared awk code. Another approach to library inclusion is to use the igawk program, which is part of the gawk distribution. Options must precede filenames and ordinary *var=value* assignments.

If no filenames are specified on the command line, awk reads from standard input.

The -- option is special: it indicates that there are no further command-line options for awk itself. Any following options are then available to your program.

The -F option redefines the default field separator, and it is conventional to make it the first command-line option. Its *fs* argument is a regular expression that immediately follows the -F, or is supplied as the next argument. The field separator can also be set with an assignment to the built-in variable FS (see Table 9-3 in "Scalar Variables," later in this chapter):

```
awk -F '\t' '{ ... }' files FS="[\f\v]" files
```

Here, the value set with the -F option applies to the first group of files, and the value assigned to FS applies to the second group.

Initializations with -v options must precede any program given directly on the command line; they take effect before the program is started, and before any files are processed. A -v option after a command-line program is interpreted as a (probably nonexistent) filename.

Initializations elsewhere on the command line are done as the arguments are processed, and may be interspersed with filenames. For example:

```
awk '{...}' Pass=1 *.tex Pass=2 *.tex
```

processes the list of files twice, once with Pass set to one and a second time with it set to two.

Initializations with string values need not be quoted unless the shell requires such quoting to protect special characters or whitespace.

The special filename - (hyphen) represents standard input. Most modern awk implementations, but not POSIX, also recognize the special name /dev/stdin for standard input, even when the host operating system does not support that filename. Similarly, /dev/stderr and /dev/stdout are available for use within awk programs to refer to standard error and standard output.

9.2 The awk Programming Model

awk views an input stream as a collection of *records*, each of which can be further subdivided into *fields*. Normally, a record is a line, and a field is a word of one or more nonwhitespace characters. However, what constitutes a record and a field is entirely under the control of the programmer, and their definitions can even be changed during processing.

An awk program consists of pairs of patterns and braced actions, possibly supplemented by functions that implement the details of the actions. For each pattern that matches the input, the action is executed, and all patterns are examined for every input record.

Either part of a pattern/action pair may be omitted. If the pattern is omitted, the action is applied to every input record. If the action is omitted, the default action is to print the matching record on standard output. Here is the typical layout of an awk program:

```
pattern  { action }        Run action if pattern matches
pattern                    Print record if pattern matches
         { action }        Run action for every record
```

Input is switched automatically from one input file to the next, and awk itself normally handles the opening, reading, and closing of each input file, allowing the user

program to concentrate on record processing. The code details are presented later in "Patterns and Actions" [9.5].

Although the patterns are often numeric or string expressions, `awk` also provides two special patterns with the reserved words `BEGIN` and `END`.

The action associated with `BEGIN` is performed just once, *before* any command-line files or ordinary command-line assignments are processed, but *after* any leading `-v` option assignments have been done. It is normally used to handle any special initialization tasks required by the program.

The `END` action is performed just once, *after* all of the input data has been processed. It is normally used to produce summary reports or to perform cleanup actions.

`BEGIN` and `END` patterns may occur in any order, anywhere in the `awk` program. However, it is conventional to make the `BEGIN` pattern the first one in the program, and to make the `END` pattern the last one.

When multiple `BEGIN` or `END` patterns are specified, they are processed in their order in the `awk` program. This allows library code included with extra `-f` options to have startup and cleanup actions.

9.3 Program Elements

Like most scripting languages, `awk` deals with numbers and strings. It provides *scalar* and *array* variables to hold data, numeric and string expressions, and a handful of statement types to process data: assignments, comments, conditionals, functions, input, loops, and output. Many features of `awk` expressions and statements are purposely similar to ones in the C programming language.

9.3.1 Comments and Whitespace

Comments in `awk` run from sharp (`#`) to end-of-line, just like comments in the shell. Blank lines are equivalent to empty comments.

Wherever whitespace is permitted in the language, any number of whitespace characters may be used, so blank lines and indentation can be used for improved readability. However, single statements usually cannot be split across multiple lines, unless the line breaks are immediately preceded with a backslash.

9.3.2 Strings and String Expressions

String constants in `awk` are delimited by quotation marks: `"This is a string constant"`. Character strings may contain any 8-bit character *except* the control character NUL (character value 0), which serves as a string terminator in the underlying

implementation language, C. The GNU implementation, gawk, removes that restriction, so gawk can safely process arbitrary binary files.

awk strings contain zero or more characters, and there is no limit, other than available memory, on the length of a string. Assignment of a string expression to a variable automatically creates a string, and the memory occupied by any previous string value of the variable is automatically reclaimed.

Backslash escape sequences allow representation of unprintable characters, just like those for the echo command shown in "Simple Output with echo" [2.5.3]. "A\tZ" contains the characters A, tab, and Z, and "\001" and "\x01" each contain just the character Ctrl-A.

Hexadecimal escape sequences are not supported by echo, but were added to awk implementations after they were introduced in the 1989 ISO C Standard. Unlike octal escape sequences, which use at most three digits, the hexadecimal escape consumes all following hexadecimal digits. gawk and nawk follow the C Standard, but mawk does not: it collects at most two hexadecimal digits, reducing "\x404142" to "@4142" instead of to the 8-bit value 0x42 = 66, which is the position of "B" in the ASCII character set. POSIX awk does not support hexadecimal escapes at all.

awk provides several convenient built-in functions for operating on strings; we treat them in detail in "String Functions" [9.9]. For now, we mention only the string-length function: length(*string*) returns the number of characters in *string*.

Strings are compared with the conventional relational operators: == (equality), != (inequality), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to). Comparison returns 0 for false and 1 for true. When strings of different lengths are compared and one string is an initial substring of the other, the shorter is defined to be less than the longer: thus, "A" < "AA" evaluates to true.

Unlike most programming languages with string datatypes, awk has no special string concatenation operator. Instead, two strings in succession are automatically concatenated. Each of these assignments sets the scalar variable s to the same four-character string:

```
s = "ABCD"
s = "AB" "CD"
s = "A" "BC" "D"
s = "A" "B" "C" "D"
```

The strings need not be constants: if we follow the last assignment with:

```
t = s s s
```

then t has the value "ABCDABCDABCD".

Conversion of a number to a string is done implicitly by concatenating the number to an empty string: n = 123, followed by s = "" n, assigns the value "123" to s. Some caution is called for when the number is not exactly representable: we address that

later when we show how to do formatted number-to-string conversions in "String Formatting" [9.9.8].

Much of the power of awk comes from its support of regular expressions. Two operators, ~ (matches) and !~ (does not match), make it easy to use regular expressions: `"ABC" ~ "^[A-Z]+$"` is true, because the left string contains only uppercase letters, and the right regular expression matches any string of (ASCII) uppercase letters. awk supports Extended Regular Expressions (EREs), as described in "Extended Regular Expressions" [3.2.3].

Regular expression constants can be delimited by either quotes or slashes: `"ABC" ~ /^[A-Z]+$/` is equivalent to the last example. Which of them to use is largely a matter of programmer taste, although the slashed form is usually preferred, since it emphasizes that the enclosed material is a regular expression, rather than an arbitrary string. However, in the rare cases where a slash delimiter might be confused with a division operator, use the quoted form.

Just as a literal quote in a quoted string must be protected by a backslash (`"...\"..."`), so must a literal slash in a slash-delimited regular expression (`/...\/.../`). When a literal backslash is needed in a regular expression, it too must be protected, but the quoted form requires an extra level of protection: `"\\\\TeX"` and `/\\TeX/` are regular expressions that each match a string containing `\TeX`.

9.3.3 Numbers and Numeric Expressions

All numbers in awk are represented as double-precision floating-point values, and we provide some of the details in the nearby sidebar. Although you do not have to become an expert in floating-point arithmetic, it is important to be aware of the limitations of computer arithmetic so that you do not expect more than the computer can deliver, and so that you can avoid some of the pitfalls.

Floating-point numbers may include a trailing power-of-10 exponent represented by the letter `e` (or `E`) and an optionally signed integer. For example, `0.03125`, `3.125e-2`, `3125e-5`, and `0.003125E1` are equivalent representations of the value 1/32. Because all arithmetic in awk is floating-point arithmetic, the expression `1/32` can be written that way without fear that it will evaluate to zero, as happens in programming languages with integer datatypes.

There is no function for explicit conversion of a string to a number, but the awk idiom is simple: just add zero to the string. For example, `s = "123"`, followed by `n = 0 + s`, assigns the number 123 to `n`.

Non-numeric strings are coerced to numbers by converting as much of the string that looks like a number: `"+123ABC"` converts to 123, and `"ABC"`, `"ABC123"`, and `""` all convert to 0.

More on Floating-Point Arithmetic

Virtually all platforms today conform to the 1985 *IEEE 754 Standard for Binary Floating-Point Arithmetic*. That standard defines a 32-bit single-precision format, a 64-bit double-precision format, and an optional extended-precision format, which is usually implemented in 80 or 128 bits. awk implementations use the 64-bit format (corresponding to the C datatype `double`), although in the interests of portability, the awk language specification is intentionally vague about the details. The POSIX awk specification says only that the arithmetic shall follow the ISO C Standard, which does not require any particular floating-point architecture.

IEEE 754 64-bit double-precision values have a sign bit, an 11-bit biased exponent, and a 53-bit significand whose leading bit is not stored. This permits representing numbers with up to about 16 decimal digits. The largest finite magnitude is about 10^{+308}, and the smallest normalized nonzero magnitude is about 10^{-308}. Most IEEE 754 implementations also support subnormal numbers, which extend the range down to about 10^{-324}, but with a loss of precision: this *gradual underflow* to zero has several desirable numerical properties, but is usually irrelevant to nonnumerical software.

Because the sign bit is explicitly represented, IEEE 754 arithmetic supports both positive and negative zero. Many programming languages get this wrong, however, and awk is no exception: some implementations print a negative zero without its minus sign.

IEEE 754 arithmetic also includes two special values, Infinity and not-a-number (NaN). Both can be signed, but the sign of NaN is not significant. They are intended to allow nonstop computation on high-performance computers while still being able to record the occurrence of exceptional conditions. When a value is too big to represent, it is said to *overflow*, and the result is Infinity. When a value is not well-defined, such as Infinity – Infinity, or 0/0, the result is a NaN.

Infinity and NaN propagate in computations: Infinity + Infinity and Infinity * Infinity produce Infinity, and NaN combined with anything produces NaN.

Infinities of the same sign compare equal. NaN compares unequal to itself: the test `(x != x)` is true only if x is a NaN.

awk was developed before IEEE 754 arithmetic became widely available, so the language does not fully support Infinity and NaN. In particular, current awk implementations trap attempts to divide by zero, even though that operation is perfectly well-defined in IEEE 754 arithmetic.

The limited precision of floating-point numbers means that some values cannot be represented exactly: the order of evaluation is significant (floating-point arithmetic is not associative), and computed results are normally rounded to the nearest representable number.

The limited range of floating-point numbers means that very small or very large numbers are not representable. On modern systems, such values are converted to zero and infinity.

Even though all numeric computations in awk are done in floating-point arithmetic, integer values can be represented exactly, provided that they are not too large. With IEEE 754 arithmetic, the 53-bit significand limits integers to at most 2^{53} = 9,007,199,254,740,992. That number is large enough that few text processing applications that involve counting things are likely to reach it.

Numeric operators in awk are similar to those in several other programming languages. We collect them in Table 9-2.

Table 9-2. Numeric operators in awk (in decreasing precedence)

Operator	Description
`++ --`	Increment and decrement (either prefix or postfix)
`^ **`	Exponentiate (right-associative)
`! + -`	Not, unary plus, unary minus
`* / %`	Multiply, divide, remainder
`+ -`	Add, subtract
`< <= == <= != > >=`	Compare
`&&`	Logical AND (short-circuit)
`||`	Logical OR (short-circuit)
`? :`	Ternary conditional
`= += -= *= /= %= ^= **=`	Assign (right-associative)

Like most programming languages, awk allows parentheses to control evaluation order. Few people can reliably remember operator precedence, especially if they work with multiple languages: when in doubt, parenthesize!

The increment and decrement operators work like those in the shell, described in "Arithmetic Expansion" [6.1.3]. In isolation, `n++` and `++n` are equivalent. However, because they have the *side effect* of updating the variable as well as returning a value, ambiguities in evaluation order can arise when they are used more than once in the same statement. For example, the result of an expression like `n++ + ++n` is implementation defined. Despite such ambiguities, the increment and decrement operators receive wide use in programming languages that have them.

Exponentiation raises the left operand to the power given by the right operand. Thus, `n^3` and `n**3` both mean the cube of `n`. The two operator names are equivalent, but come from different ancestor languages. C programmers should note that awk's ^ operator is different from C's, despite the similarity of major parts of awk and C.

Exponentiation and assignment are the only operators in awk that are *right-associative*: thus, a^b^c^d means a^(b^(c^d)), whereas a/b/c/d means ((a/b)/c)/d. These associativity rules are common to most other programming languages, and are conventional in mathematics.

In the original awk specification, the result of the remainder operator is implementation-defined when either operand is negative. POSIX awk requires that it behave like the ISO Standard C function fmod( ). This in turn requires that if x % y is representable, then the expression has the sign of x, and magnitude less than y. All awk implementations that we tested follow the POSIX mandate.

Just as in the shell, the logical operators && and || are short-circuiting forms of AND and OR: they evaluate their righthand operand only if needed.

The operator in the next-to-last row in the table is the ternary short-circuiting conditional operator. If the first operand is nonzero (true), the result is the second operand; otherwise, it is the third operand. Only one of the second and third operands is evaluated. Thus, in awk, you can write a compact assignment a = (u > w) ? x^3 : y^7 that in other programming languages might require something like this:

```
if (u > w) then
    a = x^3
else
    a = y^7
endif
```

The assignment operators are perhaps unusual for two reasons. First, the compound ones, like /=, use the left operand as the first operand on the right: n /= 3 is simply shorthand for n = n / 3. Second, the result of an assignment is an expression that may be used as part of another expression: a = b = c = 123 first assigns 123 to c (because the assignment operator is right-associative), then assigns the value of c to b, and finally, assigns the value of b to a. The result, as expected, is that a, b, and c all receive the value 123. Similarly, x = (y = 123) + (z = 321) sets x, y, and z to 444, 123, and 321, respectively.

The ** and **= operators are not part of POSIX awk and are not recognized by mawk. They should therefore be avoided in new code: use ^ and ^= instead.

Be sure to note the difference between assignment with =, and equality test with ==. Because assignments are valid expressions, the expression (r = s) ? t : u is syntactically correct, but is probably not what you intended. It assigns s to r, and then if that value is nonzero, it returns t, and otherwise returns u. This warning also applies to C, C++, Java, and other languages with = and == operators.

The built-in function int( ) returns the integer part of its argument: int(-3.14159) evaluates to –3.

awk provides some of the common elementary mathematical functions that may be familiar to you from calculators and from other programming languages: sqrt( ), sin( ), cos( ), log( ), exp( ), and so on. They are summarized in "Numeric Functions" [9.10]."

9.3.4 Scalar Variables

Variables that hold a single value are called scalar variables. In awk, as in most scripting languages, variables are not explicitly declared. Instead, they are created automatically at their first use in the program, usually by assignment of a value, which can be either a number or a string. When a variable is used, the context makes it clear whether a number or a string is expected, and the value is automatically converted from one to the other as needed.

All awk variables are created with an initial empty string value that is treated as zero when a numeric value is required.

awk variable names begin with an ASCII letter or underscore, and optionally continue with letters, underscores, and digits. Thus, variable names match the regular expression **[A-Za-z_][A-Za-z_0-9]***. There is no practical limit on the length of a variable name.

awk variable names are case-sensitive: foo, Foo, and FOO are distinct names. A common, and recommended, convention is to name local variables in lowercase, global variables with an initial uppercase letter, and built-in variables in uppercase.

awk provides several built-in variables, all spelled in uppercase. The important ones that we often need for simple programs are shown in Table 9-3.

Table 9-3. Commonly used built-in scalar variables in awk

Variable	Description
FILENAME	Name of the current input file
FNR	Record number in the current input file
FS	Field separator (regular expression) (default: " ")
NF	Number of fields in current record
NR	Record number in the job
OFS	Output field separator (default: " ")
ORS	Output record separator (default: "\n")
RS	Input record separator (regular expression in gawk and mawk only) (default: "\n")

9.3.5 Array Variables

Array variables in awk follow the same naming conventions as scalar variables, but contain zero or more data items, selected by an array index following the name.

Most programming languages require arrays to be indexed by simple integer expressions, but awk allows array indices to be arbitrary numeric or string expressions, enclosed in square brackets after the array name. If you have not encountered such arrays before, they may seem rather curious, but awk code like this fragment of an office-directory program makes their utility obvious:

```
telephone["Alice"] = "555-0134"
telephone["Bob"]   = "555-0135"
telephone["Carol"] = "555-0136"
telephone["Don"]   = "555-0141"
```

Arrays with arbitrary indices are called *associative arrays* because they associate names with values, much like humans do. Importantly, the technique that awk uses to implement these arrays allows *find*, *insert*, and *remove* operations to be done in essentially constant time, independent of the number of items stored.

Arrays in awk require neither declaration nor allocation: array storage grows automatically as new elements are referenced. Array storage is *sparse*: only those elements that are explicitly referenced are allocated. This means that you can follow `x[1] = 3.14159` with `x[10000000] = "ten million"`, without filling in elements 2 through 9999999. Most programming languages with arrays require all elements to be of the same type, but that is not the case with awk arrays.

Storage can be reclaimed when elements are no longer needed. `delete` *`array[index]`* removes an element from an array, and recent awk implementations allow `delete` *`array`* to delete all elements. We describe another way to delete array elements at the end of "String Splitting" [9.9.6].

A variable cannot be used as both a scalar and an array at the same time. Applying the `delete` statement removes *elements* of an array, but not its *name*: therefore, code like this:

```
x[1] = 123
delete x
x = 789
```

causes awk to complain that you cannot assign a value to an array name.

Sometimes, multiple indices are needed to uniquely locate tabular data. For example, the post office uses house number, street, and postal code to identify mail-delivery locations. A row/column pair suffices to identify a position in a two-dimensional grid, such as a chessboard. Bibliographies usually record author, title, edition, publisher, and year to identify a particular book. A clerk needs a manufacturer, style, color, and size to retrieve the correct pair of shoes from a stockroom.

awk simulates arrays with multiple indices by treating a *comma-separated list of indices* as a single string. However, because commas might well occur in the index values themselves, awk replaces the index-separator commas by an unprintable string stored in the built-in variable SUBSEP. POSIX says that its value is implementation-defined; generally, its default value is "\034" (the ASCII field-separator control character, FS), but you can change it if you need that string in the index values. Thus, when you write maildrop[53, "Oak Lane", "T4Q 7XV"], awk converts the index list to the string expression "53" SUBSEP "Oak Lane" SUBSEP "T4Q 7XV", and uses its string value as the index. This scheme can be subverted, although we do not recommend that you do so—these statements all print the same item:

```
print maildrop[53, "Oak Lane", "T4Q 7XV"]
print maildrop["53" SUBSEP "Oak Lane" SUBSEP "T4Q 7XV"]
print maildrop["53\034Oak Lane", "T4Q 7XV"]
print maildrop["53\034Oak Lane\034T4Q 7XV"]
```

Clearly, if you later change the value of SUBSEP, you will invalidate the indices of already-stored data, so SUBSEP really should be set just once per program, in the BEGIN action.

You can solve an astonishingly large number of data processing problems with associative arrays, once you rearrange your thinking appropriately. For a simple programming language like awk, they have shown themselves to be a superb design choice.

9.3.6 Command-Line Arguments

awk's automated handling of the command line means that few awk programs need concern themselves with it. This is quite different from the C, C++, Java, and shell worlds, where programmers are used to handling command-line arguments explicitly.

awk makes the command-line arguments available via the built-in variables ARGC (argument count) and ARGV (argument vector, or argument values). Here is a short program to illustrate their use:

```
$ cat showargs.awk
BEGIN {
    print "ARGC =", ARGC
    for (k = 0; k < ARGC; k++)
        print "ARGV[" k "] = [" ARGV[k] "]"
}
```

Here is what it produces for the general awk command line:

```
$ awk -v One=1 -v Two=2 -f showargs.awk Three=3 file1 Four=4 file2 file3
ARGC = 6
ARGV[0] = [awk]
ARGV[1] = [Three=3]
ARGV[2] = [file1]
ARGV[3] = [Four=4]
ARGV[4] = [file2]
```

```
ARGV[5] = [file3]
```

As in C and C++, the arguments are stored in array entries 0, 1, ..., ARGC – 1, and the zeroth entry is the name of the awk program itself. However, arguments associated with the -f and -v options are not available. Similarly, any command-line program is not available:

```
$ awk 'BEGIN { for (k = 0; k < ARGC; k++)
>      print "ARGV[" k "] = [" ARGV[k] "]" }' a b c
ARGV[0] = [awk]
ARGV[1] = [a]
ARGV[2] = [b]
ARGV[3] = [c]
```

Whether a directory path in the program name is visible or not is implementation-dependent:

```
$ /usr/local/bin/gawk 'BEGIN { print ARGV[0] }'
gawk

$ /usr/local/bin/mawk 'BEGIN { print ARGV[0] }'
mawk

$ /usr/local/bin/nawk 'BEGIN { print ARGV[0] }'
/usr/local/bin/nawk
```

The awk program can modify ARGC and ARGV, although it is rarely necessary to do so. If an element of ARGV is (re)set to an empty string, or deleted, awk ignores it, instead of treating it as a filename. If you eliminate trailing entries of ARGV, be sure to decrement ARGC accordingly.

awk stops interpreting arguments as options as soon as it has seen either an argument containing the program text, or the special -- option. Any following arguments that look like options must be handled by your program and then deleted from ARGV, or set to an empty string.

It is often convenient to wrap the awk invocation in a shell script. To keep the script more readable, store a lengthy program in a shell variable. You can also generalize the script to allow the awk implementation to be chosen at runtime by an environment variable with a default of nawk:

```
#! /bin/sh -
AWK=${AWK:-nawk}
AWKPROG='
   ... long program here ...
'
$AWK "$AWKPROG" "$@"
```

Single quotes protect the program text from shell interpretation, but more care is needed if the program itself contains single quotes. A useful alternative to storing the program in a shell variable is to put it in a separate file in a shared library directory that is found relative to the directory where the script is stored:

```
#! /bin/sh -
AWK=${AWK:-nawk}
$AWK -f `dirname $0`/../share/lib/myprog.awk -- "$@"
```

The `dirname` command was described in "Automating Software Builds" [8.2]. For example, if the script is in `/usr/local/bin`, then the program is in `/usr/local/share/lib`. The use of `dirname` here ensures that the script will work as long as the relative location of the two files is preserved.

9.3.7 Environment Variables

`awk` provides access to all of the environment variables as entries in the built-in array `ENVIRON`:

```
$ awk 'BEGIN { print ENVIRON["HOME"]; print ENVIRON["USER"] }'
/home/jones
jones
```

There is nothing special about the `ENVIRON` array: you can add, delete, and modify entries as needed. However, POSIX requires that subprocesses inherit the environment in effect when `awk` was started, and we found no current implementations that propagate changes to the `ENVIRON` array to either subprocesses or built-in functions. In particular, this means that you cannot control the possibly locale-dependent behavior of string functions, like `tolower( )`, with changes to `ENVIRON["LC_ALL"]`. You should therefore consider `ENVIRON` to be a read-only array.

If you need to control the locale of a subprocess, you can do so by setting a suitable environment variable in the command string. For example, you can sort a file in a Spanish locale like this:

```
system("env LC_ALL=es_ES sort infile > outfile")
```

The `system( )` function is described later, in "Running External Programs" [9.7.8].

9.4 Records and Fields

Each iteration of the implicit loop over the input files in `awk`'s programming model processes a single *record*, typically a line of text. Records are further divided into smaller strings, called *fields*.

9.4.1 Record Separators

Although records are normally text lines separated by newline characters, `awk` allows more generality through the record-separator built-in variable, `RS`.

In traditional and POSIX `awk`, `RS` must be either a single literal character, such as newline (its default value), or an empty string. The latter is treated specially: records are then paragraphs separated by one or more blank lines, and empty lines at the

start or end of a file are ignored. Fields are then separated by newlines or whatever FS is set to.

gawk and mawk provide an important extension: RS may be a regular expression, provided that it is longer than a single character. Thus, RS = "+" matches a literal plus, whereas RS = ":+" matches one or more colons. This provides much more powerful record specification, which we exploit in some of the examples in "One-Line Programs in awk" [9.6].

With a regular expression record separator, the text that matches the separator can no longer be determined from the value of RS. gawk provides it as a language extension in the built-in variable RT, but mawk does not.

Without the extension of RS to regular expressions, it can be hard to simulate regular expressions as record separators, if they can match across line boundaries, because most Unix text processing tools deal with a line at a time. Sometimes, you can use tr to convert newline into an otherwise unused character, making the data stream one giant line. However, that often runs afoul of buffer-size limits in other tools. gawk, mawk, and emacs are unusual in freeing you from the limiting view of line-oriented data.

9.4.2 Field Separators

Fields are separated from each other by strings that match the current value of the field-separator regular expression, available in the built-in variable FS.

The default value of FS, a single space, receives special interpretation: it means one or more whitespace characters (space or tab), and leading and trailing whitespace on the line is ignored. Thus, the input lines:

```
alpha beta gamma
    alpha     beta      gamma
```

both look the same to an awk program with the default setting of FS: three fields with values "alpha", "beta", and "gamma". This is particularly convenient for input prepared by humans.

For those rare occasions when a single space separates fields, simply set FS = "[ ]" to match exactly one space. With that setting, leading and trailing whitespace is no longer ignored. These two examples report different numbers of fields (two spaces begin and end the input record):

```
$ echo '  un deux trois  ' | awk -F' ' '{ print NF ":" $0 }'
3:  un deux trois

$ echo '  un deux trois  ' | awk -F'[ ]' '{ print NF ":" $0 }'
7:  un deux trois
```

The second example sees seven fields: "", "", "un", "deux", "trois", "", and "".

FS is treated as a regular expression only when it contains more than one character. FS = "." uses a period as the field separator; it is *not* a regular expression that matches any single character.

Modern awk implementations also permit FS to be an empty string. Each *character* is then a separate field, but in older implementations, each record then has only one field. POSIX says only that the behavior for an empty field separator is unspecified.

9.4.3 Fields

Fields are available to the awk program as the special names $1, $2, $3, ..., $NF. Field references need not be constant, and they are converted (by truncation) to integer values if necessary: assuming that k is 3, the values $k, $(1+2), $(27/9), $3.14159, $"3.14159", and $3 all refer to the third field.

The special field name $0 refers to the current record, initially exactly as read from the input stream, and the record separator is not part of the record. References to field numbers above the range 0 to NF are *not* erroneous: they return empty strings and do not create new fields, unless you assign them a value. References to fractional, or non-numeric, field numbers are implementation-defined. References to negative field numbers are fatal errors in all implementations that we tested. POSIX says only that references to anything other than non-negative integer field numbers are unspecified.

Fields can be assigned too, just like normal variables. For example, $1 = "alef" is legal, but has an important side effect: if the complete record is subsequently referenced, it is reassembled from the current values of the fields, but separated by the string given by the output-field-separator built-in variable, OFS, which defaults to a single space.

9.5 Patterns and Actions

Patterns and actions form the heart of awk programming. It is awk's unconventional *data-driven* programming model that makes it so attractive and contributes to the brevity of many awk programs.

9.5.1 Patterns

Patterns are constructed from string and/or numeric expressions: when they evaluate to nonzero (true) for the current input record, the associated action is carried out. If a pattern is a bare regular expression, then it means to match the entire input record against that expression, as if you had written $0 ~ /regexp/ instead of just /regexp/. Here are some examples to give the general flavor of selection patterns:

```
NF == 0          Select empty records
NF > 3           Select records with more than 3 fields
NR < 5           Select records 1 through 4
```

```
(FNR == 3) && (FILENAME ~ /[.][ch]$/)    Select record 3 in C source files
$1 ~ /jones/                             Select records with "jones" in field 1
/[Xx][Mm][Ll]/                           Select records containing "XML", ignoring lettercase
$0 ~ /[Xx][Mm][Ll]/                      Same as preceding selection
```

awk adds even more power to the matching by permitting *range expressions*. Two expressions separated by a comma select records from one matching the left expression up to, and including, the record that matches the right expression. If both range expressions match a record, the selection consists of that single record. This behavior is different from that of sed, which looks for the range end only in records that follow the start-of-range record. Here are some examples:

```
(FNR == 3), (FNR == 10)                        Select records 3 through 10 in each input file
/<[Hh][Tt][Mm][Ll]>/, /<\/[Hh][Tt][Mm][Ll]>/   Select body of an HTML document
/[aeiouy][aeiouy]/, /[^aeiouy][^aeiouy]/       Select from two vowels to two nonvowels
```

In the BEGIN action, FILENAME, FNR, NF, and NR are initially undefined; references to them return a null string or zero.

If a program consists only of actions with BEGIN patterns, awk exits after completing the last action, without reading any files.

On entry to the first END action, FILENAME is the name of the last input file processed, and FNR, NF, and NR retain their values from the last input record. The value of $0 in the END action is unreliable: gawk and mawk retain it, nawk does not, and POSIX is silent.

9.5.2 Actions

We have now covered most of the awk language elements needed to select records. The action section that optionally follows a pattern is, well, where the action is: it specifies how to process the record.

awk has several statement types that allow construction of arbitrary programs. However, we delay presentation of most of them until "Statements" [9.7]. For now, apart from the assignment statement, we consider only the simple print statement.

In its simplest form, a bare print means to print the current input record ($0) on standard output, followed by the value of the output record separator, ORS, which is by default a single newline character. These programs are therefore equivalent:

```
1                          Pattern is true, default action is to print
NR > 0  { print }          Print when have records, is always true
1       { print }          Pattern is true, explicit print, default value
        { print }          No pattern is treated as true, explicit print, default value
        { print $0 }       Same, but with explicit value to print
```

A one-line awk program that contained any of those lines would simply copy the input stream to standard output.

More generally, a print statement can contain zero or more comma-separated expressions. Each is evaluated, converted to a string if necessary, and output on standard output, separated by the value of the output field separator, OFS. The last item is followed by the value of the output record separator, ORS.

The argument lists for print and its companions printf and sprintf (see "String Formatting" [9.9.8]) may optionally be parenthesized. The parentheses eliminate a parsing ambiguity when the argument list contains a relational operator, since < and > are also used in I/O redirection, as described in "User-Controlled Input" [9.7.6] and "Output Redirection" [9.7.7].

Here are some complete awk program examples. In each, we print just the first three input fields, and by omitting the selection pattern, we select all records. Semicolons separate awk program statements, and we vary the action code slightly to change the output field separators:

```
$ echo 'one two three four' | awk '{ print $1, $2, $3 }'
one two three

$ echo 'one two three four' | awk '{ OFS = "..."; print $1, $2, $3 }'
one...two...three

$ echo 'one two three four' | awk '{ OFS = "\n"; print $1, $2, $3 }'
one
two
three
```

Changing the output field separator without assigning any field does *not* alter $0:

```
$ echo 'one two three four' | awk '{ OFS = "\n"; print $0 }'
one two three four
```

However, if we change the output field separator, and we assign at least one of the fields (even if we do not change its value), then we force reassembly of the record with the new field separator:

```
$ echo 'one two three four' | awk '{ OFS = "\n"; $1 = $1; print $0 }'
one
two
three
four
```

9.6 One-Line Programs in awk

We have now covered enough awk to do useful things with as little as one line of code; few other programming languages can do so much with so little. In this section, we present some examples of these one-liners, although page-width limitations sometimes force us to wrap them onto more than one line. In some of the examples, we show multiple ways to program a solution in awk, or with other Unix tools:

- We start with a simple implementation in awk of the Unix word-count utility, wc:

```
awk '{ C += length($0) + 1; W += NF } END { print NR, W, C }'
```

Notice that pattern/action groups need not be separated by newlines, even though we usually do that for readability. Although we could have included an initialization block of the form `BEGIN { C = W = 0 }`, awk's guaranteed default initializations make it unnecessary. The character count in `C` is updated at each record to count the record length, plus the newline that is the default record separator. The word count in `W` accumulates the number of fields. We do not need to keep a line-count variable because the built-in record count, `NR`, automatically tracks that information for us. The `END` action handles the printing of the one-line report that `wc` produces.

- awk exits immediately without reading any input if its program is empty, so it can match cat as an efficient data sink:

```
$ time cat *.xml > /dev/null
0.035u 0.121s 0:00.21 71.4%     0+0k 0+0io 99pf+0w
$ time awk '' *.xml
0.136u 0.051s 0:00.21 85.7%     0+0k 0+0io 140pf+0w
```

Apart from issues with NUL characters, awk can easily emulate cat—these two examples produce identical output:

```
cat *.xml
awk 1 *.xml
```

- To print original data values and their logarithms for one-column datafiles, use this:

```
awk '{ print $1, log($1) }' file(s)
```

- To print a random sample of about 5 percent of the lines from text files, use the pseudorandom-number generator function (see "Numeric Functions" [9.10]), which produces a result uniformly distributed between zero and one:

```
awk 'rand( ) < 0.05' file(s)
```

- Reporting the sum of the *n*-th column in tables with whitespace-separated columns is easy:

```
awk -v COLUMN=n '{ sum += $COLUMN } END { print sum }' file(s)
```

- A minor tweak instead reports the average of column *n*:

```
awk -v COLUMN=n '{ sum += $COLUMN } END { print sum / NR }' file(s)
```

- To print the running total for expense files whose records contain a description and an amount in the last field, use the built-in variable `NF` in the computation of the total:

```
awk '{ sum += $NF; print $0, sum }' file(s)
```

- Here are three ways to search for text in files:

```
egrep 'pattern|pattern' file(s)
awk '/pattern|pattern/' file(s)
awk '/pattern|pattern/ { print FILENAME ":" FNR ":" $0 }' file(s)
```

- If you want to restrict the search to just lines 100–150, you can use two tools and a pipeline, albeit with loss of location information:

```
sed -n -e 100,150p -s file(s) | egrep 'pattern'
```

We need GNU sed here for its -s option, which restarts line numbering for each file. Alternatively, you can use awk with a fancier pattern:

```
awk '(100 <= FNR) && (FNR <= 150) && /pattern/ \
      { print FILENAME ":" FNR ":" $0 }' file(s)
```

- To swap the second and third columns in a four-column table, assuming tab separators, use any of these:

```
awk -F'\t' -v OFS='\t' '{ print $1, $3, $2, $4 }' old > new
awk 'BEGIN { FS = OFS = "\t" } { print $1, $3, $2, $4 }' old > new
awk -F'\t' '{ print $1 "\t" $3 "\t" $2 "\t" $4 }' old > new
```

- To convert column separators from tab (shown here as •) to ampersand, use either of these:

```
sed -e 's/•/\&/g' file(s)
awk 'BEGIN { FS = "\t"; OFS = "&" } { $1 = $1; print }' file(s)
```

- Both of these pipelines eliminate duplicate lines from a sorted stream:

```
sort file(s) | uniq
sort file(s) | awk 'Last != $0 { print } { Last = $0 }'
```

- To convert carriage-return/newline line terminators to newline terminators, use one of these:

```
sed -e 's/\r$//' file(s)
sed -e 's/^M$//' file(s)
mawk 'BEGIN { RS = "\r\n" } { print }' file(s)
```

The first sed example needs a modern version that recognizes escape sequences. In the second example, ^M represents a literal Ctrl-M (carriage return) character. For the third example, we need either gawk or mawk because nawk and POSIX awk do not support more than a single character in RS.

- To convert single-spaced text lines to double-spaced lines, use any of these:

```
sed -e 's/$/\n/' file(s)
awk 'BEGIN { ORS = "\n\n" } { print }' file(s)
awk 'BEGIN { ORS = "\n\n" } 1' file(s)
awk '{ print $0 "\n" }' file(s)
awk '{ print; print "" }' file(s)
```

As before, we need a modern sed version. Notice how a simple change to the output record separator, ORS, in the first awk example solves the problem: the rest of the program just prints each record. The two other awk solutions require more processing for each record, and usually are slower than the first one.

- Conversion of double-spaced lines to single spacing is equally easy:

```
gawk 'BEGIN { RS="\n *\n" } { print }' file(s)
```

- To locate lines in Fortran 77 programs that exceed the 72-character line-length limit,* either of these does the job:

```
egrep -n '^.{73,}' *.f
awk 'length($0) > 72 { print FILENAME ":" FNR ":" $0 }' *.f
```

We need a POSIX-compliant egrep for the extended regular expression that matches 73 or more of any character.

- To extract properly hyphenated International Standard Book Number (ISBN) values from documents, we need a lengthy, but straightforward, regular expression, with the record separator set to match all characters that cannot be part of an ISBN:

```
gawk 'BEGIN { RS = "[^0-9Xx]" }
/[0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9]-[0-9Xx]/' \
    file(s)
```

  With a POSIX-conformant awk, that long regular expression can be shortened to /[0-9][-0-9]{10}-[-0-9Xx]/. Our tests found that gawk --posix, HP/Compaq/DEC OSF/1 awk, Hewlett-Packard HP-UX awk, IBM AIX awk, and Sun Solaris /usr/xpg4/bin/awk are the only ones that support the POSIX extension of braced interval expressions in regular expressions.

- To strip angle-bracketed markup tags from HTML documents, treat the tags as record separators, like this:

```
mawk 'BEGIN { ORS = " "; RS = "<[^<>]*>" } { print }' *.html
```

  By setting ORS to a space, HTML markup gets converted to a space, and all input line breaks are preserved.

- Here is how we can extract all of the titles from a collection of XML documents, such as the files for this book, and print them, one title per line, with surrounding markup. This program works correctly even when the titles span multiple lines, and handles the uncommon, but legal, case of spaces between the tag word and the closing angle bracket:

```
$ mawk -v ORS=' ' -v RS='[ \n]' '/<title *>/, /<\/title *>/' *.xml |
>      sed -e 's@</title *> *@&\n@g'
...
<title>Enough awk to Be Dangerous</title>
<title>Freely available awk versions</title>
<title>The awk Command Line</title>
...
```

  The awk program produces a single line of output, so the modern sed filter supplies the needed line breaks. We could eliminate sed here, but to do so, we need some awk statements discussed in the next section.

* The Fortran line-length limit was not a problem in the old days of punched cards, but once screen-based editing became common, it became a source of nasty bugs caused by the compiler's silently ignoring statement text beyond column 72.

9.7 Statements

Programming languages need to support sequential, conditional, and iterative execution. awk provides these features with statements borrowed largely from the C programming language. This section also covers the different statement types that are specific to awk.

9.7.1 Sequential Execution

Sequential execution is provided by lists of statements, written one per line, or separated by semicolons. The three lines:

```
n = 123
s = "ABC"
t = s n
```

can also be written like this:

```
n = 123; s = "ABC"; t = s n
```

In one-liners, we often need the semicolon form, but in awk programs supplied from files, we usually put each statement on its own line, and we rarely need a semicolon.

Wherever a single statement is expected, a *compound statement* consisting of a braced group of statements can be used instead. Thus, the actions associated with awk patterns are just compound statements.

9.7.2 Conditional Execution

awk provides for conditional execution with the `if` statement:

```
if (expression)
    statement1

if (expression)
    statement1
else
    statement2
```

If the *expression* is nonzero (true), then execute *statement1*. Otherwise, if there is an `else` part, execute *statement2*. Each of these statements may themselves be `if` statements, so the general form of a multibranch conditional statement is usually written like this:

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
...
```

```
else if (expressionk)
    statementk
else
    statementk+1
```

The optional final `else` is always associated with the closest preceding `if` at the same level.

In a multibranch `if` statement, the conditional expressions are tested in order: the first one that matches selects the associated statement for execution, after which control continues with the statement following the complete `if` statement, without evaluating conditional expressions in the remainder of the statement. If no expressions match, then the final `else` branch, if present, is selected.

9.7.3 Iterative Execution

`awk` provides four kinds of iterative statements (loops):

- Loop with a termination test at the beginning:

```
while (expression)
    statement
```

- Loop with a termination test at the end:

```
do
    statement
while (expression)
```

- Loop a countable number of times:

```
for (expr1; expr2; expr3)
    statement
```

- Loop over elements of an associative array:

```
for (key in array)
    statement
```

The `while` loop satisfies many iteration needs, typified by *while we have data, process it*. The `do` loop is much less common: it appears, for example, in optimization problems that reduce to *compute an error estimate, and repeat while the error is too big*. Both loop while the expression is nonzero (true). If the expression is initially zero, then the `while` loop body is not executed at all, whereas the `do` loop body is executed just once.

The first form of the `for` loop contains three semicolon-separated expressions, any or all of which may be empty. The first expression is evaluated before the loop begins. The second is evaluated at the start of each iteration, and while it is nonzero (true), the loop continues. The third is evaluated at the end of each iteration. The traditional loop from 1 to *n* is written like this:

```
for (k = 1; k <= n; k++)
    statement
```

However, the index need not increase by one each iteration. The loop can be run backward like this:

```
for (k = n; k >= 1; k--)
    statement
```

Because floating-point arithmetic is usually inexact, avoid `for`-statement expressions that evaluate to nonintegral values. For example, the loop:

```
$ awk 'BEGIN { for (x = 0; x <= 1; x += 0.05) print x }'
...
0.85
0.9
0.95
```

does not print 1 in its last iteration because the additions of the inexactly represented value 0.05 produce a final `x` value that is slightly larger than 1.0.

C programmers should note that `awk` lacks a comma operator, so the three `for` loop expressions cannot be comma-separated lists of expressions.

The second form of the `for` loop is used for iterating over the elements of an array when the number of elements is not known, or do not form a computable integer sequence. The elements are selected in arbitrary order, so the output of:

```
for (name in telephone)
    print name "\t" telephone[name]
```

is unlikely to be in the order that you want. We show how to solve that problem in "Output Redirection" [9.7.7]. The `split( )` function, described in "String Splitting" [9.9.6], handles the case of multiply-indexed arrays.

As in the shell, the `break` statement exits the innermost loop prematurely:

```
for (name in telephone)
    if (telephone[name] == "555-0136")
        break
print name, "has telephone number 555-0136"
```

However, the shell-style multilevel `break` *n* statement is not supported.

Just like in the shell, the `continue` statement jumps to the end of the loop body, ready for the next iteration. `awk` does not recognize the shell's multilevel `continue` *n* statement. To illustrate the `continue` statement, the program in Example 9-1 determines by brute-force testing of divisors whether a number is composite or prime (recall that a prime number is any whole number larger than one that has no integral divisors other than one and itself), and prints any factorization that it can find.

Example 9-1. Integer factorization

```
# Compute integer factorizations of integers supplied one per line.
# Usage:
```

Example 9-1. Integer factorization (continued)

```
#       awk -f factorize.awk
{
    n = int($1)
    m = n = (n >= 2) ? n : 2
    factors = ""
    for (k = 2; (m > 1) && (k^2 <= n); )
    {
        if (int(m % k) != 0)
        {
            k++
            continue
        }
        m /= k
        factors = (factors == "") ? ("" k) : (factors " * " k)
    }
    if ((1 < m) && (m < n))
        factors = factors " * " m
    print n, (factors == "") ? "is prime" : ("= " factors)
}
```

Notice that the loop variable `k` is incremented, and the `continue` statement executed, only when we find that `k` is *not* a divisor of `m`, so the third expression in the `for` statement is empty.

If we run it with suitable test input, we get this output:

```
$ awk -f factorize.awk test.dat
2147483540 = 2 * 2 * 5 * 107374177
2147483541 = 3 * 7 * 102261121
2147483542 = 2 * 3137 * 342283
2147483543 is prime
2147483544 = 2 * 2 * 2 * 3 * 79 * 1132639
2147483545 = 5 * 429496709
2147483546 = 2 * 13 * 8969 * 9209
2147483547 = 3 * 3 * 11 * 21691753
2147483548 = 2 * 2 * 7 * 76695841
2147483549 is prime
2147483550 = 2 * 3 * 5 * 5 * 19 * 23 * 181 * 181
```

9.7.4 Array Membership Testing

The membership test *key* `in` *array* is an expression that evaluates to 1 (true) if *key* is an index element of *array*. The test can be inverted with the *not* operator: `!(`*key* `in` *array*`)` is 1 if *key* is not an index element of *array*; the parentheses are mandatory.

For arrays with multiple subscripts, use a parenthesized comma-separated list of subscripts in the test: `(`*i*`, `*j*`, …, `*n*`)` `in` *array*.

A membership test never creates an array element, whereas referencing an element always creates it, if it does not already exist. Thus, you should write:

```
if ("Sally" in telephone)
    print "Sally is in the directory"
```

rather than:

```
if (telephone["Sally"] != "")
    print "Sally is in the directory"
```

because the second form installs her in the directory with an empty telephone number, if she is not already there.

It is important to distinguish finding an *index* from finding a particular *value*. The index membership test requires constant time, whereas a search for a value takes time proportional to the number of elements in the array, illustrated by the `for` loop in the `break` statement example in the previous section. If you need to do both of these operations frequently, it is worthwhile to construct an inverted-index array:

```
for (name in telephone)
    name_by_telephone[telephone[name]] = name
```

You can then use `name_by_telephone["555-0136"]` to find `"Carol"` in constant time. Of course, this assumes that all values are unique: if two people share a telephone, the `name_by_telephone` array records only the last name stored. You can solve that problem with just a bit more code:

```
for (name in telephone)
{
    if (telephone[name] in name_by_telephone)
        name_by_telephone[telephone[name]] = \
            name_by_telephone[telephone[name]] "\t" name
    else
        name_by_telephone[telephone[name]] = name
}
```

Now `name_by_telephone` contains tab-separated lists of people with the same telephone number.

9.7.5 Other Control Flow Statements

We have already discussed the `break` and `continue` statements for interrupting the control flow in iterative statements. Sometimes, you need to alter the control flow in `awk`'s matching of input records against the patterns in the list of pattern/action pairs. There are three cases to handle:

Skip further pattern checking for this record only
: Use the `next` statement. Some implementations do not permit `next` in user-defined functions (described in "User-Defined Functions" [9.8]).

Skip further pattern checking for the current input file
: `gawk` and recent releases of `nawk` provide the `nextfile` statement. It causes the current input file to be closed immediately, and pattern matching restarts with records from the next file on the command line.

You can easily simulate the nextfile statement in older awk implementation, with some loss of efficiency. Replace the nextfile statement with SKIPFILE = FILENAME; next, and then add these new pattern/action pairs at the beginning of the program:

```
FNR == 1                { SKIPFILE = "" }
FILENAME == SKIPFILE    { next }
```

The first pattern/action pair resets SKIPFILE to an empty string at the start of each file so that the program works properly if the same filename appears as two successive arguments. Even though records continue to be read from the current file, they are immediately ignored by the next statement. When end-of-file is reached and the next input file is opened, the second pattern no longer matches, so the next statement in its action is not executed.

Skip further execution of the entire job, and return a status code to the shell

Use the exit *n* statement.

9.7.6 User-Controlled Input

awk's transparent handling of input files specified on the command line means that most awk programs never have to open and process files themselves. It is quite possible to do so, however, through awk's getline statement. For example, a spellchecker usually needs to load in one or more dictionaries before it can do its work.

getline returns a value and can be used like a function, even though it is actually a statement, and one with somewhat unconventional syntax. The return value is +1 when input has been successfully read, 0 at end-of-file, and –1 on error. It can be used in several different ways that are summarized in Table 9-4.

Table 9-4. getline variations

Syntax	Description
getline	Read the next record from the current input file into $0, and update NF, NR, and FNR.
getline *var*	Read the next record from the current input file into *var*, and update NR and FNR.
getline < *file*	Read the next record from *file* into $0, and update NF.
getline *var* < *file*	Read the next record from *file* into *var*.
cmd | getline	Read the next record from the external command, *cmd*, into $0, and update NF.
cmd | getline *var*	Read the next record from the external command, *cmd*, into *var*.

Let's look at some of these uses of getline. First, we pose a question, and then read and check the answer:

```
print "What is the square root of 625?"
getline answer
print "Your reply, ", answer ", is", (answer == 25) ? "right." : "wrong."
```

If we wanted to ensure that input came from the controlling terminal, rather than standard input, we instead could have used:

```
getline answer < "/dev/tty"
```

Next, we load a list of words from a dictionary:

```
nwords = 1
while ((getline words[nwords] < "/usr/dict/words") > 0)
    nwords++
```

Command pipelines are a powerful feature in awk. The pipeline is specified in a character string, and can contain arbitrary shell commands. It is used with getline like this:

```
"date" | getline now
close("date")
print "The current time is", now
```

Most systems limit the number of open files, so when we are through with the pipeline, we use the close( ) function to close the pipeline file. In older awk implementations, close was a statement, so there is no portable way to use it like a function and get a reliable return code back.

Here is how you can use a command pipeline in a loop:

```
command = "head -n 15 /etc/hosts"
while ((command | getline s) > 0)
    print s
close(command)
```

We used a variable to hold the pipeline to avoid repetition of a possibly complicated string, and to ensure that all uses of the command match exactly. In command strings, every character is significant, and even an inadvertent difference of a single space would refer to a different command.

9.7.7 Output Redirection

The print and printf statements (see "String Formatting" [9.9.8]) normally send their output to standard output. However, the output can be sent to a file instead:

```
print "Hello, world" > file
printf("The tenth power of %d is %d\n", 2, 2^10) > "/dev/tty"
```

To append to an existing file (or create a new one if it does not yet exist), use >> output redirection:

```
print "Hello, world" >> file
```

You can use output redirection to the same file on any number of output statements. When you are finished writing output, use close(*file*) to close the file and free its resources.

Avoid mixing > and >> for the same file without an intervening close( ). In awk, these operators tell how the output file should be opened. Once open, the file remains open until it is explicitly closed, or until the program terminates. Contrast that behavior with the shell, where redirection requires the file to be opened and closed at each command.

Alternatively, you can send output to a pipeline:

```
for (name in telephone)
    print name "\t" telephone[name] | "sort"
close("sort")
```

As with input from a pipeline, close an output pipeline as soon as you are through with it. This is particularly important if you need to read the output in the same program. For example, you can direct the output to a temporary file, and then read it after it is complete:

```
tmpfile = "/tmp/telephone.tmp"
command = "sort > " tmpfile
for (name in telephone)
    print name "\t" telephone[name] | command
close(command)
while ((getline < tmpfile) > 0)
    print
close(tmpfile)
```

Pipelines in awk put the entire Unix toolbox at our disposal, eliminating the need for much of the library support offered in other programming languages, and helping to keep the language small. For example, awk does not provide a built-in function for sorting because it would just duplicate functionality already available in the powerful sort command described in "Sorting Text" [4.1].

Recent awk implementations, but not POSIX, provide a function to flush buffered data to the output stream: fflush(*file*). Notice the doubled initial ff (for *file flush*). It returns 0 on success and –1 on failure. The behavior of calls to fflush( ) (omitted argument) and fflush("") (empty string argument) is implementation-dependent: avoid such uses in portable programs.

9.7.8 Running External Programs

We showed earlier how the getline statement and output redirection in awk pipelines can communicate with external programs. The system(*command*) function provides a third way: its return value is the exit status code of the command. It first flushes any buffered output, then starts an instance of /bin/sh, and sends it the command. The shell's standard error and standard output are the same as that of the awk program, so unless the command's I/O is redirected, output from both the awk program and the shell command appears in the expected order.

Here is a shorter solution to the telephone-directory sorting problem, using a temporary file and system( ) instead of an awk pipeline:

```
tmpfile = "/tmp/telephone.tmp"
for (name in telephone)
    print name "\t" telephone[name] > tmpfile
close(tmpfile)
system("sort < " tmpfile)
```

The temporary file must be closed before the call to system( ) to ensure that any buffered output is properly recorded in the file.

There is no need to call close( ) for commands run by system( ), because close( ) is only for files or pipes opened with the I/O redirection operators and getline, print, or printf.

The system( ) function provides an easy way to remove the script's temporary file:

```
system("rm -f " tmpfile)
```

The command passed to system( ) can contain multiple lines:

```
system("cat <<EOFILE\nuno\ndos\ntres\nEOFILE")
```

It produces the output expected when copying the here document to standard output:

```
uno
dos
tres
```

Because each call to system( ) starts a fresh shell, there is no simple way to pass data between commands in separate calls to system( ), other than via intermediate files. There is an easy solution to this problem—use an output pipeline to the shell to send multiple commands:

```
shell = "/usr/local/bin/ksh"
print "export INPUTFILE=/var/tmp/myfile.in" | shell
print "export OUTPUTFILE=/var/tmp/myfile.out" | shell
print "env | grep PUTFILE" | shell
close(shell)
```

This approach has the added virtue that you get to choose the shell, but has the drawback that you cannot portably retrieve the exit-status value.

9.8 User-Defined Functions

The awk statements that we have covered so far are sufficient to write almost any data processing program. Because human programmers are poor at understanding large blocks of code, we need a way to split such blocks into manageable chunks that each perform an identifiable job. Most programming languages provide this ability,

through features variously called functions, methods, modules, packages, and subroutines. For simplicity, awk provides only functions. As in C, awk functions can optionally return a scalar value. Only a function's documentation, or its code, if quite short, can make clear whether the caller should expect a returned value.

Functions can be defined anywhere in the program at top level: before, between, or after pattern/action groups. In single-file programs, it is conventional to place all functions after the pattern/action code, and it is usually most convenient to keep them in alphabetical order. awk does not care about these conventions, but people do.

A function definition looks like this:

```
function name(arg1, arg2, ..., argn)
{
    statement(s)
}
```

The named arguments are used as local variables within the function body, and they hide any global variables of the same name. The function may be used elsewhere in the program by calls of the form:

```
name(expr1, expr2, ..., exprn)              Ignore any return value

result = name(expr1, expr2, ..., exprn)     Save return value in result
```

The expressions at the point of each call provide initial values for the function-argument variables. The parenthesized argument list must immediately follow the function name, without any intervening whitespace.

Changes made to scalar arguments are not visible to the caller, but changes made to arrays *are* visible. In other words, scalars are passed *by value*, whereas arrays are passed *by reference*: the same is true of the C language.

A `return` *`expression`* statement in the function body terminates execution of the body, and returns control to the point of the call, with the value of *`expression`*. If *`expression`* is omitted, then the returned value is implementation-defined. All of the systems that we tested returned either a numeric zero, or an empty string. POSIX does not address the issue of a missing `return` statement or value.

All variables used in the function body that do not occur in the argument list are *global*. awk permits a function to be called with fewer arguments than declared in the function definition; the extra arguments then serve as *local* variables. Such variables are commonly needed, so it is conventional to list them in the function argument list, prefixed by some extra whitespace, as shown in Example 9-2. Like all other variables in awk, the extra arguments are initialized to an empty string at function entry.

Example 9-2. Searching an array for a value

```
function find_key(array, value,        key)
{
    # Search array[ ] for value, and return key such that
```

Example 9-2. Searching an array for a value (continued)

```
    # array[key] == value, or return "" if value is not found

    for (key in array)
        if (array[key] == value)
            return key
    return ""
}
```

Failure to list local variables as extra function arguments leads to hard-to-find bugs when they clash with variables used in calling code. gawk provides the `--dump-variables` option to help you check for this.

As in most programming languages, awk functions can call themselves: this is known as *recursion*. Obviously, the programmer must make some provision for eventual termination: this is usually done by making the job smaller for each successive invocation so that at some point, no further recursion is needed. Example 9-3 shows a famous example from elementary number theory that uses a method credited to the Greek mathematician Euclid (ca. 300 BCE), but probably known at least 200 years earlier, to find the greatest common denominator of two integers.

Example 9-3. Euclid's greatest common denominator algorithm

```
function gcd(x, y,        r)
{
    # return the greatest common denominator of integer x, y

    x = int(x)
    y = int(y)
    # print x, y
    r = x % y
    return (r == 0) ? y : gcd(y, r)
}
```

If we add this action

```
    { g = gcd($1, $2); print "gcd(" $1 ", " $2 ") =", g }
```

to the code in Example 9-3 and then we uncomment the `print` statement and run it from a file, we can see how the recursion works:

```
$ echo 25770 30972 | awk -f gcd.awk
25770 30972
30972 25770
25770 5202
5202 4962
4962 240
240 162
162 78
78 6
gcd(25770, 30972) = 6
```

Euclid's algorithm always takes relatively few steps, so there is no danger of overflowing the *call stack* inside awk that keeps track of the nested function-call history. However, that is not always the case. There is a particularly nasty function discovered by the German mathematician Wilhelm Ackermann* in 1926 whose value, and recursion depth, grow much faster than exponentially. It can be defined in awk with the code in Example 9-4.

Example 9-4. Ackermann's worse-than-exponential function

```
function ack(a, b)
{
    N++                         # count recursion depth
    if (a == 0)
        return (b + 1)
    else if (b == 0)
        return (ack(a - 1, 1))
    else
        return (ack(a - 1, ack(a, b - 1)))
}
```

If we augment it with a test action:

```
{ N = 0; print "ack(" $1 ", " $2 ") = ", ack($1, $2), "[" N " calls]" }
```

and run it from a test file, we find:

```
$ echo 2 2 | awk -f ackermann.awk
ack(2, 2) =  7 [27 calls]

$ echo 3 3 | awk -f ackermann.awk
ack(3, 3) =  61 [2432 calls]

$ echo 3 4 | awk -f ackermann.awk
ack(3, 4) =  125 [10307 calls]

$ echo 3 8 | awk -f ackermann.awk
ack(3, 8) =  2045 [2785999 calls]
```

`ack(4, 4)` is completely uncomputable.

9.9 String Functions

In "Strings and String Expressions" [9.3.2] we introduced the `length(`*`string`*`)` function, which returns the length of a string *`string`*. Other common string operations include concatenation, data formatting, lettercase conversion, matching, searching, splitting, string substitution, and substring extraction.

* See *http://mathworld.wolfram.com/AckermannFunction.html* for background and history of the Ackermann function.

9.9.1 Substring Extraction

The substring function, substr(*string*, *start*, *len*), returns a copy of the substring of *len* characters from *string* starting from character *start*. Character positions are numbered starting from one: substr("abcde", 2, 3) returns "bcd". The *len* argument can be omitted, in which case, it defaults to length(*string*) - *start* + 1, selecting the remainder of the string.

It is *not* an error for the arguments of substr( ) to be out of bounds, but the result may be implementation-dependent. For example, nawk and gawk evaluate substr("ABC", -3, 2) as "AB", whereas mawk produces the empty string "". All of them produce an empty string for substr("ABC", 4, 2) and for substr("ABC", 1, 0). gawk's --lint option diagnoses out-of-bounds arguments in substr( ) calls.

9.9.2 Lettercase Conversion

Some alphabets have uppercase and lowercase forms of each letter, and in string searching and matching, it is often desirable to ignore case differences. awk provides two functions for this purpose: tolower(*string*) returns a copy of *string* with all characters replaced by their lowercase equivalents, and toupper(*string*) returns a copy with uppercase equivalents. Thus, tolower("aBcDeF123") returns "abcdef123", and toupper("aBcDeF123") returns "ABCDEF123". These functions are fine for ASCII letters, but they do not correctly case-convert accented letters. Nor do they handle unusual situations, like the German lowercase letter ß (eszett, sharp s), whose uppercase form is two letters, SS.

9.9.3 String Searching

index(*string*, *find*) searches the text in *string* for the string *find*. It returns the starting position of *find* in *string*, or 0 if *find* is not found in *string*. For example, index("abcdef", "de") returns 4.

Subject to the caveats noted in "Lettercase Conversion" [9.9.2], you can make string searches ignore lettercase like this: index(tolower(*string*), tolower(*find*)). Because case insensitivity is sometimes needed in an entire program, gawk provides a useful extension: set the built-in variable IGNORECASE to nonzero to ignore lettercase in string matches, searches, and comparisons.

index( ) finds the first occurrence of a substring, but sometimes, you want to find the last occurrence. There is no standard function to do that, but we can easily write one, shown in Example 9-5.

Example 9-5. Reverse string search

```
function rindex(string, find,        k, ns, nf)
{
    # Return index of last occurrence of find in string,
```

Example 9-5. Reverse string search (continued)

```
    # or 0 if not found

    ns = length(string)
    nf = length(find)
    for (k = ns + 1 - nf; k >= 1; k--)
        if (substr(string, k, nf) == find)
            return k
    return 0
}
```

The loop starts at a `k` value that lines up the ends of the strings `string` and `find`, extracts a substring from `string` that is the same length as `find`, and compares that substring with `find`. If they match, then `k` is the desired index of the last occurrence, and the function returns that value. Otherwise, we back up one character, terminating the loop when `k` moves past the beginning of `string`. When that happens, `find` is known not to be found in `string`, and we return an index of 0.

9.9.4 String Matching

`match(`*`string`*`,` *`regexp`*`)` matches *`string`* against the regular expression *`regexp`*, and returns the index in *`string`* of the match, or 0 if there is no match. This provides more information than the expression (*`string`* `~` *`regexp`*), which evaluates to either 1 or 0. In addition, `match( )` has a useful side effect: it sets the global variables `RSTART` to the index in *`string`* of the start of the match, and `RLENGTH` to the length of the match. The matching substring is then available as `substr(`*`string`*`, RSTART, RLENGTH)`.

9.9.5 String Substitution

`awk` provides two functions for string substitution: `sub(`*`regexp, replacement, target`*`)` and `gsub(`*`regexp, replacement, target`*`)`. `sub( )` matches *`target`* against the regular expression *`regexp`*, and replaces the leftmost longest match by the string *`replacement`*. `gsub( )` works similarly, but replaces all matches (the prefix `g` stands for *global*). Both functions return the number of substitutions. If the third argument is omitted, it defaults to the current record, `$0`. These functions are unusual in that they modify their scalar arguments: consequently, they cannot be written in the `awk` language itself. For example, a check-writing application might use `gsub(/[^$-0-9.,]/, "*", amount)` to replace with asterisks all characters other than those that can legally appear in the amount.

In a call to `sub(`*`regexp, replacement, target`*`)` or `gsub(`*`regexp, replacement, target`*`)`, each instance of the character `&` in *`replacement`* is replaced in *`target`* by the text matched by *`regexp`*. Use `\&` to disable this feature, and remember to double the backslash if you use it in a quoted string. For example, `gsub(/[aeiouyAEIOUY]/, "&&")` doubles all vowels in the current record, `$0`, whereas `gsub(/[aeiouyAEIOUY]/, "\\&\\&")` replaces each vowel by a pair of ampersands.

gawk provides a more powerful generalized-substitution function, gensub( ); see the *gawk*(1) manual pages for details.

Substitution is often a better choice for data reduction than indexing and substring operations. Consider the problem of extracting the string value from an assignment in a file with text like this:

```
composer =      "P. D. Q. Bach"
```

With substitution, we can use:

```
value = $0
sub(/^ *[a-z]+ *= *"/, "", value)
sub(/" *$/, "", value)
```

whereas with indexing using code like this:

```
start = index($0, "\"") + 1
end = start - 1 + index(substr($0, start), "\"")
value = substr($0, start, end - start)
```

we need to count characters rather carefully, we do not match the data pattern as precisely, and we have to create two substrings.

9.9.6 String Splitting

The convenient splitting into fields $1, $2, ..., $NF that awk automatically provides for the current input record, $0, is also available as a function: split(*string*, *array*, *regexp*) breaks *string* into pieces stored in successive elements of *array*, where the pieces lie between substrings matched by the regular expression *regexp*. If *regexp* is omitted, then the current value of the built-in field-separator variable, FS, is used. The function return value is the number of elements in *array*. Example 9-6 demonstrates split( ).

Example 9-6. Test program for field splitting

```
{
    print "\nField separator = FS = \"" FS "\""
    n = split($0, parts)
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""

    print "\nField separator = \"[ ]\""
    n = split($0, parts, "[ ]")
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""

    print "\nField separator = \":\""
    n = split($0, parts, ":")
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""
```

Example 9-6. Test program for field splitting (continued)

```
    print ""
}
```

If we put the test program shown in Example 9-6 into a file and run it interactively, we can see how `split( )` works:

```
$ awk -f split.awk
  Harold  and Maude

Field separator = FS = " "
parts[1] = "Harold"
parts[2] = "and"
parts[3] = "Maude"

Field separator = "[ ]"
parts[1] = ""
parts[2] = ""
parts[3] = "Harold"
parts[4] = ""
parts[5] = "and"
parts[6] = "Maude"

Field separator = :
parts[1] = "  Harold  and Maude"

root:x:0:1:The Omnipotent Super User:/root:/sbin/sh

Field separator = FS = " "
parts[1] = "root:x:0:1:The"
parts[2] = "Omnipotent"
parts[3] = "Super"
parts[4] = "User:/root:/sbin/sh"

Field separator = "[ ]"
parts[1] = "root:x:0:1:The"
parts[2] = "Omnipotent"
parts[3] = "Super"
parts[4] = "User:/root:/sbin/sh"

Field separator = ":"
parts[1] = "root"
parts[2] = "x"
parts[3] = "0"
parts[4] = "1"
parts[5] = "The Omnipotent Super User"
parts[6] = "/root"
parts[7] = "/sbin/sh"
```

Notice the difference between the default field-separator value of `" "`, which causes leading and trailing whitespace to be ignored and runs of whitespace to be treated as a single space, and a field-separator value of `"[ ]"`, which matches exactly one space. For most text processing applications, the first of these gives the desired behavior.

The colon field-separator example shows that `split( )` produces a one-element array when the field separator is not matched, and demonstrates splitting of a record from a typical Unix administrative file, `/etc/passwd`.

Recent `awk` implementations provide a useful generalization: `split(string, chars, "")` breaks *`string`* apart into one-character elements in `chars[1]`, `chars[2]`, ..., `chars[length(string)]`. Older implementations require less efficient code like this:

```
n = length(string)
for (k = 1; k <= n; k++)
    chars[k] = substr(string, k, 1)
```

The call `split("", array)` deletes all elements in *`array`*: it is a faster method for array element deletion than the loop:

```
for (key in array)
    delete array[key]
```

when `delete` *`array`* is not supported by your `awk` implementation.

`split( )` is an essential function for iterating through multiply subscripted arrays in `awk`. Here is an example:

```
for (triple in maildrop)
{
    split(triple, parts, SUBSEP)
    house_number = parts[1]
    street = parts[2]
    postal_code = parts[3]
    ...
}
```

9.9.7 String Reconstruction

There is no standard built-in `awk` function that is the inverse of `split( )`, but it is easy to write one, as shown in Example 9-7. `join( )` ensures that the argument array is not referenced unless the index is known to be in bounds. Otherwise, a call with a zero array length might create `array[1]`, modifying the caller's array. The inserted field separator is an ordinary string, rather than a regular expression, so for general regular expressions passed to `split( )`, `join( )` does not reconstruct the original string exactly.

Example 9-7. Joining array elements into a string

```
function join(array, n, fs,        k, s)
{
    # Recombine array[1]...array[n] into a string, with elements
    # separated by fs

    if (n >= 1)
    {
        s = array[1]
```

Example 9-7. Joining array elements into a string (continued)

```
        for (k = 2; k <= n; k++)
            s = s fs array[k]
    }
    return (s)
}
```

9.9.8 String Formatting

The last string functions that we present format numbers and strings under user control: `sprintf(`*`format,expression1,expression2,...`*`)` returns the formatted string as its function value. `printf( )` works the same way, except that it prints the formatted string on standard output or redirected to a file, instead of returning it as a function value. Newer programming languages replace format control strings with potentially more powerful formatting functions, but at a significant increase in code verbosity. For typical text processing applications, `sprintf( )` and `printf( )` are nearly always sufficient.

`printf( )` and `sprintf( )` format strings are similar to those of the shell `printf` command that we described in "The Full Story on printf" [7.4]. We summarize the `awk` format items in Table 9-5. These items can each be augmented by the same field width, precision, and flag modifiers discussed in Chapter 7.

The `%i`, `%u`, and `%X` items were not part of the 1987 language redesign, but modern implementations support them. Despite the similarity with the shell `printf` command, `awk`'s handling of the `%c` format item differs for integer arguments, and output with `%u` for negative arguments may disagree because of differences in shell and `awk` arithmetic.

Table 9-5. printf and sprintf format specifiers

Item	Description
`%c`	ASCII character. Print the first character of the corresponding string argument, or the character whose number in the host character set is the corresponding integer argument, usually taken modulo 256.
`%d`, `%i`	Decimal integer.
`%e`	Floating-point format ([-]*d.precisione*[+-]*dd*).
`%f`	Floating-point format ([-]*ddd.precision*).
`%g`	`%e` or `%f` conversion, whichever is shorter, with trailing zeros removed.
`%o`	Unsigned octal value.
`%s`	String.
`%u`	Unsigned value. `awk` numbers are floating-point values: small negative integer values are output as large positive ones because the sign bit is interpreted as a data bit.
`%x`	Unsigned hexadecimal number. Letters `a`–`f` represent 10 to 15.
`%X`	Unsigned hexadecimal number. Letters `A`–`F` represent 10 to 15.
`%%`	Literal `%`.

Most of the format items are straightforward. However, we caution that *accurate* conversion of binary floating-point values to decimal strings, and the reverse, is a surprisingly difficult problem whose proper solution was only found in about 1990, and can require very high intermediate precision. awk implementations generally use the underlying C library for the conversions required by sprintf( ) format items, and although library quality continues to improve, there are still platforms in which the accuracy of floating-point conversions is deficient. In addition, differences in floating-point hardware and instruction evaluation order mean that floating-point results from almost any programming language vary slightly across different architectures.

When floating-point numbers appear in print statements, awk formats them according to the value of the built-in variable OFMT, which defaults to "%.6g". You can redefine OFMT as needed.

Similarly, when floating-point numbers are converted to strings by concatenation, awk formats them according to the value of another built-in variable, CONVFMT.* Its default value is also "%.6g".

The test program in Example 9-8 produces output like this with a recent nawk version on a Sun Solaris SPARC system:

```
$ nawk -f ofmt.awk
[ 1] OFMT = "%.6g"      123.457
[ 2] OFMT = "%d"        123
[ 3] OFMT = "%e"        1.234568e+02
[ 4] OFMT = "%f"        123.456789
[ 5] OFMT = "%g"        123.457
[ 6] OFMT = "%25.16e"      1.2345678901234568e+02
[ 7] OFMT = "%25.16f"        123.4567890123456806
[ 8] OFMT = "%25.16g"           123.4567890123457
[ 9] OFMT = "%25d"                            123
[10] OFMT = "%.25d"     0000000000000000000000123
[11] OFMT = "%25d"                     2147483647
[12] OFMT = "%25d"                     2147483647    Expected 2147483648
[13] OFMT = "%25d"                     2147483647    Expected 9007199254740991
[14] OFMT = "%25.0f"             9007199254740991
```

Evidently, despite the availability of 53-bit precision in floating-point values, on this platform nawk caps them at 32-bit integer limits for %d formats. Slightly different values were produced by runs of the same nawk version on other architectures. Example 9-8 shows the source for ofmt.awk.

Example 9-8. Testing the effect of OFMT

```
BEGIN {
    test( 1, OFMT,      123.4567890123456789)
```

* Originally, OFMT served for both output and string conversions, but POSIX introduced CONVFMT to separate their quite distinct purposes. Most implementations now have both, but SGI IRIX and Sun Solaris /usr/bin/nawk lack CONVFMT.

Example 9-8. Testing the effect of OFMT (continued)

```
    test( 2, "%d",      123.4567890123456789)
    test( 3, "%e",      123.4567890123456789)
    test( 4, "%f",      123.4567890123456789)
    test( 5, "%g",      123.4567890123456789)
    test( 6, "%25.16e", 123.4567890123456789)
    test( 7, "%25.16f", 123.4567890123456789)
    test( 8, "%25.16g", 123.4567890123456789)
    test( 9, "%25d",    123.4567890123456789)
    test(10, "%.25d",   123.4567890123456789)
    test(11, "%25d",    2^31 - 1)
    test(12, "%25d",    2^31)
    test(13, "%25d",    2^52 + (2^52 - 1))
    test(14, "%25.0f",  2^52 + (2^52 - 1))
}

function test(n,fmt,value,    save_fmt)
{
    save_fmt = OFMT
    OFMT = fmt
    printf("[%2d] OFMT = \"%s\"\t", n, OFMT)
    print value
    OFMT = save_fmt
}
```

We found that output for this test was quite sensitive to particular awk implementations, and even different releases of the same one. For example, with gawk, we get:

```
$ gawk -f ofmt.awk
...
[11] OFMT = "%25d"      2147483647                  Expected right-adjusted result
...
[13] OFMT = "%25d"                    9.0072e+15    Expected 9007199254740991
...
```

The informal language definition in the 1987 awk book specifies the default value of OFMT, but makes no mention of the effect of other values. Perhaps in recognition of implementation differences, POSIX says that the result of conversions is unspecified if OFMT is not a floating-point format specification, so gawk's behavior here is allowed.

With mawk, we find:

```
$ mawk -f ofmt.awk
...
[ 2] OFMT = "%d"        1079958844                  Expected 123
...
[ 9] OFMT = "%25d"                    1079958844    Expected 123
[10] OFMT = "%.25d"     0000000000000001079958844   Expected 00...00123
[11] OFMT = "%25d"      2147483647                  Expected right-adjusted result
[12] OFMT = "%25d"                    1105199104    Expected 2147483648
[13] OFMT = "%25d"                    1128267775    Expected 9007199254740991
...
```

There are evidently inconsistencies and idiosyncrasies in the handling of output of large numbers with the formats `%d` and, in separate tests, `%i`. Fortunately, you can get correct output from all `awk` implementations by using a `%.0f` format instead.

9.10 Numeric Functions

`awk` provides the elementary numeric functions listed in Table 9-6. Most of them are common to many programming languages, and their accuracy depends on the quality of the underlying native mathematical-function library.

Table 9-6. Elementary numeric functions

Function	Description
`atan2(y, x)`	Return the arctangent of y/x as a value in $-\pi$ to $+\pi$.
`cos(x)`	Return the cosine of x (measured in *radians*) as a value in −1 to +1.
`exp(x)`	Return the exponential of x, e^x.
`int(x)`	Return the integer part of x, truncating toward zero.
`log(x)`	Return the natural logarithm of x.
`rand( )`	Return a uniformly distributed pseudorandom number, r, such that $0 \le r < 1$.
`sin(x)`	Return the sine of x (measured in *radians*) as a value in −1 to +1.
`sqrt(x)`	Return the square root of x.
`srand(x)`	Set the pseudorandom-number generator seed to x, and return the current seed. If x is omitted, use the current time in seconds, relative to the system epoch. If `srand( )` is not called, `awk` starts with the same default seed on each run; `mawk` does not.

The pseudorandom-number generator functions `rand( )` and `srand( )` are the area of largest variation in library functions in different `awk` implementations because some of them use native system-library functions instead of their own code, and the pseudorandom-number generating algorithms and precision vary. Most algorithms for generation of such numbers step through a sequence from a finite set without repetition, and the sequence ultimately repeats itself after a number of steps called the *period* of the generator. Library documentation sometimes does not make clear whether the unit interval endpoints, 0.0 and 1.0, are included in the range of `rand( )`, or what the period is.

The ambiguity in the generator's result interval endpoints makes programming harder. Suppose that you want to generate pseudorandom integers between 0 and 100 inclusive. If you use the simple expression `int(rand( )*100)`, you will not get the value 100 at all if `rand( )` never returns 1.0, and even if it does, you will get 100 much less frequently than any other integer between 0 and 100, since it is produced only once in the generator period, when the generator returns the exact value 1.0. Fudging by changing the multiplier from 100 to 101 does not work either because you might get an out-of-range result of 101 on some systems.

The irand( ) function in Example 9-9 provides a better solution to the problem of generating pseudorandom integers. irand( ) forces integer endpoints and then, if the requested range is empty or invalid, returns one endpoint. Otherwise, irand( ) samples an integer that might be one larger than the interval width, adds it to low, and then *retries* if the result is out of range. Now it does not matter whether rand( ) ever returns 1.0, and the return values from irand( ) are as uniformly distributed as the rand( ) values.

Example 9-9. Generating pseudorandom integers

```
function irand(low, high,        n)
{
    # Return a pseudorandom integer n such that low <= n <= high

    # Ensure integer endpoints
    low = int(low)
    high = int(high)

    # Sanity check on argument order
    if (low >= high)
        return (low)

    # Find a value in the required range
    do
        n = low + int(rand( ) * (high + 1 - low))
    while ((n < low) || (high < n))

    return (n)
}
```

In the absence of a call to srand(*x*), gawk and nawk use the same initial seed on each run so that runs are reproducible; mawk does not. Seeding with the current time via a call to srand( ) to get different sequences on each run is reasonable, *if* the clock is precise enough. Unfortunately, although machine speeds have increased dramatically, most time-of-day clocks used in current awk implementations still tick only once per second, so it is quite possible that successive runs of a simulation execute within the same clock tick. The solution is to avoid calling srand( ) more than once per run, and to introduce a delay of at least one second between runs:

```
$ for k in 1 2 3 4 5
> do
>     awk 'BEGIN {
>                     srand( )
>                     for (k = 1; k <= 5; k++)
>                         printf("%.5f ", rand( ))
>                     print ""
>                 }'
>     sleep 1
> done
0.29994 0.00751 0.57271 0.26084 0.76031
0.81381 0.52809 0.57656 0.12040 0.60115
```

```
0.32768 0.04868 0.58040 0.98001 0.44200
0.84155 0.56929 0.58422 0.83956 0.28288
0.35539 0.08985 0.58806 0.69915 0.12372
```

Without the `sleep 1` statement, the output lines are often identical.

9.11 Summary

A surprisingly large number of text processing jobs can be handled with the subset of `awk` that we have presented in this chapter. Once you understand `awk`'s command line, and how it automatically handles input files, the programming job reduces to specifying record selections and their corresponding actions. This kind of minimalist *data-driven* programming can be extremely productive. By contrast, most conventional programming languages would burden you with dozens of lines of fairly routine code to loop over a list of input files, and for each file, open the file, read, select, and process records until end-of-file, and finally, close the file.

When you see how simple it is to process records and fields with `awk`, your view of data processing can change dramatically. You begin to divide large tasks into smaller, and more manageable, ones. For example, if you are faced with processing complex binary files, such as those used for databases, fonts, graphics, slide makers, spreadsheets, typesetters, and word processors, you might design, or find, a pair of utilities to convert between the binary format and a suitably marked-up simple text format, and then write small filters in `awk` or other scripting languages to manipulate the text representation.