8 Functions

In this chapter:
- *Built-in Functions*
- *User-Defined Functions*

This chapter describes *awk*'s built-in functions, which fall into three categories: numeric, string, and I/O. *gawk* provides additional groups of functions to work with values that represent time, do bit manipulation, and internationalize and localize programs.

Besides the built-in functions, *awk* has provisions for writing new functions that the rest of a program can use. The second half of this chapter describes these *user-defined functions*.

Built-in Functions

Built-in functions are always available for your *awk* program to call. This section defines all the built-in functions in *awk*; some of these are mentioned in other sections but are summarized here for your convenience.

Calling Built-in Functions

To call one of *awk*'s built-in functions, write the name of the function followed by arguments in parentheses. For example, `atan2(y + z, 1)` is a call to the function `atan2` and has two arguments.

Whitespace is ignored between the built-in function name and the open parenthesis, and it is good practice to avoid using whitespace there. User-defined functions do not permit whitespace in this way, and it is easier to avoid mistakes by following a simple convention that always works—no whitespace after a function name.

Each built-in function accepts a certain number of arguments. In some cases, arguments can be omitted. The defaults for omitted arguments vary from function to function and are described under the individual functions. In some *awk* implementations, extra arguments given to built-in functions are ignored. However, in *gawk*, it is a fatal error to give extra arguments to a built-in function.

When a function is called, expressions that create the function's actual parameters are evaluated completely before the call is performed. For example, in the following code fragment:

```
i = 4
j = sqrt(i++)
```

the variable `i` is incremented to the value 5 before `sqrt` is called with a value of 4 for its actual parameter. The order of evaluation of the expressions used for the function's parameters is undefined. Thus, avoid writing programs that assume that parameters are evaluated from left to right or from right to left. For example:

```
i = 5
j = atan2(i++, i *= 2)
```

If the order of evaluation is left to right, then `i` first becomes 6, and then 12, and `atan2` is called with the two arguments 6 and 12. But if the order of evaluation is right to left, `i` first becomes 10, then 11, and `atan2` is called with the two arguments 11 and 10.

Numeric Functions

The following list describes all of the built-in functions that work with numbers. Optional parameters are enclosed in square brackets ([ ]):

`int(`*x*`)`
: This returns the nearest integer to *x*, located between *x* and zero and truncated toward zero.

  For example, `int(3)` is 3, `int(3.9)` is 3, `int(-3.9)` is −3, and `int(-3)` is −3 as well.

`sqrt(`*x*`)`
: This returns the positive square root of *x*. *gawk* reports an error if *x* is negative. Thus, `sqrt(4)` is 2.

`exp(`*x*`)`
: This returns the exponential of *x* (e^x) or reports an error if *x* is out of range. The range of values *x* can have depends on your machine's floating-point representation.

`log(`*x*`)`
: This returns the natural logarithm of *x*, if *x* is positive; otherwise, it reports an error.

`sin(`*x*`)`
: This returns the sine of *x*, with *x* in radians.

`cos(`*x*`)`
: This returns the cosine of *x*, with *x* in radians.

`atan2(`*y, x*`)`
: This returns the arctangent of *y* / *x* in radians.

`rand()`
: This returns a random number. The values of `rand` are uniformly distributed between zero and one. The value is never zero and never one.*

Often random integers are needed instead. Following is a user-defined function that can be used to obtain a random non-negative integer less than *n*:

```
function randint(n) {
    return int(n * rand())
}
```

The multiplication produces a random number greater than zero and less than `n`. Using `int`, this result is made into an integer between zero and `n` − 1, inclusive.

The following example uses a similar function to produce random integers between one and *n*. This program prints a new random number for each input record:

```
# Function to roll a simulated die.
function roll(n) { return 1 + int(rand() * n) }

# Roll 3 six-sided dice and
# print total number of points.
{
    printf("%d points\n",
           roll(6)+roll(6)+roll(6))
}
```

* The C version of `rand` is known to produce fairly poor sequences of random numbers. However, nothing requires that an *awk* implementation use the C `rand` to implement the *awk* version of `rand`. In fact, *gawk* uses the BSD `random` function, which is considerably better than `rand`, to produce random numbers.

In most *awk* implementations, including *gawk*, `rand` starts generating numbers from the same starting number, or *seed*, each time you run *awk*. Thus, a program generates the same results each time you run it. The numbers are random within one *awk* run but predictable from run to run. This is convenient for debugging, but if you want a program to do different things each time it is used, you must change the seed to a value that is different in each run. To do this, use `srand`.

`srand([`*x*`])`

The function `srand` sets the starting point, or seed, for generating random numbers to the value *x*.

Each seed value leads to a particular sequence of random numbers.* Thus, if the seed is set to the same value a second time, the same sequence of random numbers is produced again.

Different *awk* implementations use different random-number generators internally. Don't expect the same *awk* program to produce the same series of random numbers when executed by different versions of *awk*.

If the argument *x* is omitted, as in `srand()`, then the current date and time of day are used for a seed. This is the way to get random numbers that are truly unpredictable.

The return value of `srand` is the previous seed. This makes it easy to keep track of the seeds in case you need to consistently reproduce sequences of random numbers.

String-Manipulation Functions

The functions in this section look at or change the text of one or more strings. Optional parameters are enclosed in square brackets ([ ]). Those functions that are specific to *gawk* are marked with a pound sign (`#`):

`asort(`*source*`[, `*dest*`]) #`

`asort` is a *gawk*-specific extension, returning the number of elements in the array *source*. The contents of *source* are sorted using *gawk*'s normal rules for comparing values, and the indices of the sorted values of *source* are replaced with sequential integers starting with one. If the optional array *dest* is speci-

* Computer-generated random numbers really are not truly random. They are technically known as "pseudorandom." This means that while the numbers in a sequence appear to be random, you can in fact generate the same sequence of random numbers over and over again.

fied, then *source* is duplicated into *dest*. *dest* is then sorted, leaving the indices of *source* unchanged. For example, if the contents of a are as follows:

```
a["last"] = "de"
a["first"] = "sac"
a["middle"] = "cul"
```

A call to asort:

```
asort(a)
```

results in the following contents of a:

```
a[1] = "cul"
a[2] = "de"
a[3] = "sac"
```

The asort function is described in more detail in the section "Sorting Array Values and Indices with gawk" in Chapter 7, *Arrays in awk*. asort is a *gawk* extension; it is not available in compatibility mode (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*).

index(*in*, *find*)

This searches the string *in* for the first occurrence of the string *find*, and returns the position in characters at which that occurrence begins in the string *in*. Consider the following example:

```
$ awk 'BEGIN { print index("peanut", "an") }'
3
```

If *find* is not found, index returns zero. (Remember that string indices in *awk* start at one.)

length([*string*])

This returns the number of characters in *string*. If *string* is a number, the length of the digit string representing that number is returned. For example, length("abcde") is 5. By contrast, length(15 * 35) works out to 3. In this example, 15 * 35 = 525, and 525 is then converted to the string "525", which has three characters.

If no argument is supplied, length returns the length of $0.

In older versions of *awk*, the length function could be called without any parentheses. Doing so is marked as "deprecated" in the POSIX standard. This means that while a program can do this, it is a feature that can eventually be removed from a future version of the standard. Therefore, for programs to be maximally portable, always supply the parentheses.

`match(`*`string`*`, `*`regexp`*` [, `*`array`*`])`

The `match` function searches *string* for the longest, leftmost substring matched by the regular expression, *regexp*. It returns the character position, or *index*, at which that substring begins (one, if it starts at the beginning of *string*). If no match is found, it returns zero.

The order of the first two arguments is backwards from most other string functions that work with regular expressions, such as `sub` and `gsub`. It might help to remember that for `match`, the order is the same as for the `~` operator: *string* `~` *regexp*.

The `match` function sets the built-in variable `RSTART` to the index. It also sets the built-in variable `RLENGTH` to the length in characters of the matched substring. If no match is found, `RSTART` is set to zero, and `RLENGTH` to −1.

For example:

```
{
    if ($1 == "FIND")
        regex = $2
    else {
        where = match($0, regex)
        if (where != 0)
            print "Match of", regex, "found at",
                        where, "in", $0
    }
}
```

This program looks for lines that match the regular expression stored in the variable `regex`. This regular expression can be changed. If the first word on a line is `FIND`, `regex` is changed to be the second word on that line. Therefore, if given:

```
FIND ru+n
My program runs
but not very quickly
FIND Melvin
JF+KM
This line is property of Reality Engineering Co.
Melvin was here.
```

awk prints:

```
Match of ru+n found at 12 in My program runs
Match of Melvin found at 1 in Melvin was here.
```

If *array* is present, it is cleared, and then the 0th element of *array* is set to the entire portion of *string* matched by *regexp*. If *regexp* contains parentheses, the integer-indexed elements of *array* are set to contain the portion of *string* matching the corresponding parenthesized subexpression. For example:

```
$ echo foooobazbarrrrr |
> gawk '{ match($0, /(fo+).+(ba*r)/, arr)
>           print arr[1], arr[2] }'
foooo barrrrr
```

The *array* argument to `match` is a *gawk* extension. In compatibility mode (see the section “Command-Line Options” in Chapter 11), using a third argument is a fatal error.

`split(`*`string`*`, `*`array`*`[, `*`fieldsep`*`])`

This function divides *string* into pieces separated by *fieldsep* and stores the pieces in *array*. The first piece is stored in *`array`*`[1]`, the second piece in *`array`*`[2]`, and so forth. The string value of the third argument, *fieldsep*, is a regexp describing where to split *string* (much as `FS` can be a regexp describing where to split input records). If *fieldsep* is omitted, the value of `FS` is used. `split` returns the number of elements created. If *string* does not match *fieldsep*, *array* is empty and `split` returns zero.

The `split` function splits strings into pieces in a manner similar to the way input lines are split into fields. For example:

```
split("cul-de-sac", a, "-")
```

splits the string `cul-de-sac` into three fields using `-` as the separator. It sets the contents of the array `a` as follows:

```
a[1] = "cul"
a[2] = "de"
a[3] = "sac"
```

The value returned by this call to `split` is three.

As with input field-splitting, when the value of *fieldsep* is `" "`, leading and trailing whitespace is ignored, and the elements are separated by runs of whitespace. Also as with input field-splitting, if *fieldsep* is the null string, each individual character in the string is split into its own array element. (This is a *gawk*-specific extension.)

Modern implementations of *awk*, including *gawk*, allow the third argument to be a regexp constant (`/abc/`) as well as a string. (d.c.) The POSIX standard allows this as well.

Before splitting the string, `split` deletes any previously existing elements in the array *array*. If *string* does not match *fieldsep* at all, *array* has one element only. The value of that element is the original *string*.

`sprintf(`*`format, expression1`*`, ...)`

This returns (without printing) the string that `printf` would have printed out with the same arguments (see the section "Using printf Statements for Fancier Printing" in Chapter 4, *Printing Output*). For example:

```
pival = sprintf("pi = %.2f (approx.)", 22/7)
```

assigns the string `"pi = 3.14 (approx.)"` to the variable `pival`.

`strtonum(`*`str`*`) #`

Examines *str* and returns its numeric value. If *str* begins with a leading `0`, `strtonum` assumes that *str* is an octal number. If *str* begins with a leading `0x` or `0X`, `strtonum` assumes that *str* is a hexadecimal number. For example:

```
$ echo 0x11 | gawk '{ printf "%d\n", strtonum($1) }'
17
```

Using the `strtonum` function is *not* the same as adding zero to a string value; the automatic coercion of strings to numbers works only for decimal data, not for octal or hexadecimal.*

`strtonum` is a *gawk* extension; it is not available in compatibility mode (see the section "Command-Line Options" in Chapter 11).

`sub(`*`regexp`*`, `*`replacement`*` [, `*`target`*`])`

The `sub` function alters the value of *target*. It searches this value, which is treated as a string, for the leftmost, longest substring matched by the regular expression *regexp*. Then the entire string is changed by replacing the matched text with *replacement*. The modified string becomes the new value of *target*.

This function is peculiar because *target* is not simply used to compute a value, and not just any expression will do—it must be a variable, field, or array element so that `sub` can store a modified value there. If this argument is omitted, then the default is to use and alter `$0`. For example:

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

sets `str` to `"wither, water, everywhere"`, by replacing the leftmost longest occurrence of `at` with `ith`.

The `sub` function returns the number of substitutions made (either one or zero).

* Unless you use the *--non-decimal-data* option, which isn't recommended. See the section "Allowing Nondecimal Input Data" in Chapter 10, *Advanced Features of gawk*, for more information.

If the special character `&` appears in *replacement*, it stands for the precise substring that was matched by *regexp*. (If the regexp can match more than one string, then this precise substring may vary.) For example:

```
{ sub(/candidate/, "& and his wife"); print }
```

changes the first occurrence of `candidate` to `candidate and his wife` on each input line. Here is another example:

```
$ awk 'BEGIN {
>     str = "daabaaa"
>     sub(/a+/, "C&C", str)
>     print str
> }'
dCaaCbaaa
```

This shows how `&` can represent a nonconstant string and also illustrates the "leftmost, longest" rule in regexp matching (see the section "How Much Text Matches?" in Chapter 2, *Regular Expressions*).

The effect of this special character (`&`) can be turned off by putting a backslash before it in the string. As usual, to insert one backslash in the string, you must write two backslashes. Therefore, write `\\&` in a string constant to include a literal `&` in the replacement. For example, the following shows how to replace the first `|` on each line with an `&`:

```
{ sub(/\|/, "\\&"); print }
```

As mentioned, the third argument to `sub` must be a variable, field or array reference. Some versions of *awk* allow the third argument to be an expression that is not an *lvalue*. In such a case, `sub` still searches for the pattern and returns zero or one, but the result of the substitution (if any) is thrown away because there is no place to put it. Such versions of *awk* accept expressions such as the following:

```
sub(/USA/, "United States", "the USA and Canada")
```

For historical compatibility, *gawk* accepts erroneous code, such as in the previous example. However, using any other nonchangeable object as the third parameter causes a fatal error and your program will not run.

Finally, if the *regexp* is not a regexp constant, it is converted into a string, and then the value of that string is treated as the regexp to match.

`gsub(`*`regexp`*`, `*`replacement`*` [, `*`target`*`])`

This is similar to the `sub` function, except `gsub` replaces *all* of the longest, leftmost, *nonoverlapping* matching substrings it can find. The `g` in `gsub` stands for "global," which means replace everywhere. For example:

```
{ gsub(/Britain/, "United Kingdom"); print }
```

replaces all occurrences of the string `Britain` with `United Kingdom` for all input records.

The `gsub` function returns the number of substitutions made. If the variable to search and alter (*target*) is omitted, then the entire input record (`$0`) is used. As in `sub`, the characters `&` and `\` are special, and the third argument must be assignable.

`gensub(`*`regexp`*`, `*`replacement`*`, `*`how`*`[, `*`target`*`]) #`

`gensub` is a general substitution function. Like `sub` and `gsub`, it searches the target string *target* for matches of the regular expression *regexp*. Unlike `sub` and `gsub`, the modified string is returned as the result of the function and the original target string is *not* changed. If *how* is a string beginning with `g` or `G`, then it replaces all matches of *regexp* with *replacement*. Otherwise, *how* is treated as a number that indicates which match of *regexp* to replace. If no *target* is supplied, `$0` is used.

`gensub` provides an additional feature that is not available in `sub` or `gsub`: the ability to specify components of a regexp in the replacement text. This is done by using parentheses in the regexp to mark the components and then specifying `\`*`N`* in the replacement text, where *N* is a digit from 1 to 9. For example:

```
$ gawk '
> BEGIN {
>      a = "abc def"
>      b = gensub(/(.+) (.+)/, "\\2 \\1", "g", a)
>      print b
> }'
def abc
```

As with `sub`, you must type two backslashes in order to get one into the string. In the replacement text, the sequence `\0` represents the entire matched text, as does the character `&`.

The following example shows how you can use the third argument to control which match of the regexp should be changed:

```
$ echo a b c a b c |
> gawk '{ print gensub(/a/, "AA", 2) }'
a b c AA b c
```

In this case, `$0` is used as the default target string. `gensub` returns the new string as its result, which is passed directly to `print` for printing.

If the *how* argument is a string that does not begin with `g` or `G`, or if it is a number that is less than or equal to zero, only one substitution is performed. If *how* is zero, *gawk* issues a warning message.

If *regexp* does not match *target*, `gensub`'s return value is the original unchanged value of *target*.

`gensub` is a *gawk* extension; it is not available in compatibility mode (see the section "Command-Line Options" in Chapter 11).

`substr(`*`string`*`, `*`start`*` [, `*`length`*`])`

This returns a *length*-character-long substring of *string*, starting at character number *start*. The first character of a string is character number one.* For example, `substr("washington", 5, 3)` returns `"ing"`.

If *length* is not present, this function returns the whole suffix of *string* that begins at character number *start*. For example, `substr("washington", 5)` returns `"ington"`. The whole suffix is also returned if *length* is greater than the number of characters remaining in the string, counting from character *start*.

The string returned by `substr` *cannot* be assigned. Thus, it is a mistake to attempt to change a portion of a string, as shown in the following example:

```
string = "abcdef"
# try to get "abCDEf", won't work
substr(string, 3, 3) = "CDE"
```

It is also a mistake to use `substr` as the third argument of `sub` or `gsub`:

```
gsub(/xyz/, "pdq", substr($0, 5, 20))  # WRONG
```

(Some commercial versions of *awk* do in fact let you use `substr` this way, but doing so is not portable.)

If you need to replace bits and pieces of a string, combine `substr` with string concatenation, in the following manner:

```
string = "abcdef"
...
string = substr(string, 1, 2) "CDE" substr(string, 6)
```

`tolower(`*`string`*`)`

This returns a copy of *string*, with each uppercase character in the string replaced with its corresponding lowercase character. Nonalphabetic characters are left unchanged. For example, `tolower("MiXeD cAsE 123")` returns `"mixed case 123"`.

`toupper(`*`string`*`)`

This returns a copy of *string*, with each lowercase character in the string replaced with its corresponding uppercase character. Nonalphabetic characters are left unchanged. For example, `toupper("MiXeD cAsE 123")` returns `"MIXED CASE 123"`.

* This is different from C and C++, in which the first character is number zero.

More about \ and & with sub, gsub, and gensub

When using `sub`, `gsub`, or `gensub`, and trying to get literal backslashes and ampersands into the replacement text, you need to remember that there are several levels of *escape processing* going on.

First, there is the *lexical* level, which is when *awk* reads your program and builds an internal copy of it that can be executed. Then there is the runtime *level*, which is when *awk* actually scans the replacement string to determine what to generate.

At both levels, *awk* looks for a defined set of characters that can come after a backslash. At the lexical level, it looks for the escape sequences listed in the section "Escape Sequences" in Chapter 2. Thus, for every `\` that *awk* processes at the runtime level, type two backslashes at the lexical level. When a character that is not valid for an escape sequence follows the `\`, Unix *awk* and *gawk* both simply remove the initial `\` and put the next character into the string. Thus, for example, `"a\qb"` is treated as `"aqb"`.

At the runtime level, the various functions handle sequences of `\` and `&` differently. The situation is (sadly) somewhat complex. Historically, the `sub` and `gsub` functions treated the two character sequence `\&` specially; this sequence was replaced in the generated text with a single `&`. Any other `\` within the *replacement* string that did not precede an `&` was passed through unchanged. This is illustrated in Table 8-1.

Table 8-1. Historical Escape Sequence Processing for sub and gsub

You type	sub sees	sub generates
`\&`	`&`	The matched text
`\\&`	`\&`	A literal `&`
`\\\&`	`\&`	A literal `&`
`\\\\&`	`\\&`	A literal `\&`
`\\\\\&`	`\\&`	A literal `\&`
`\\\\\\&`	`\\\&`	A literal `\\&`
`\\q`	`\q`	A literal `\q`

Table 8-1 shows both the lexical-level processing, where an odd number of backslashes becomes an even number at the runtime level, as well as the runtime processing done by `sub`. (For the sake of simplicity, the rest of the following tables only show the case of even numbers of backslashes entered at the lexical level.)

The problem with the historical approach is that there is no way to get a literal `\` followed by the matched text.

The 1992 POSIX standard attempted to fix this problem. The standard says that `sub` and `gsub` look for either a `\` or an `&` after the `\`. If either one follows a `\`, that character is output literally. The interpretation of `\` and `&` then becomes as shown in Table 8-2.

Table 8-2. 1992 POSIX Rules for sub and gsub Escape Sequence Processing

You type	sub sees	sub generates
`&`	`&`	The matched text
`\\&`	`\&`	A literal `&`
`\\\\&`	`\\&`	A literal `\`, then the matched text
`\\\\\\&`	`\\\&`	A literal `\&`

This appears to solve the problem. Unfortunately, the phrasing of the standard is unusual. It says, in effect, that `\` turns off the special meaning of any following character, but for anything other than `\` and `&`, such special meaning is undefined. This wording leads to two problems:

- Backslashes must now be doubled in the *replacement* string, breaking historical *awk* programs.
- To make sure that an *awk* program is portable, *every* character in the *replacement* string must be preceded with a backslash.*

The POSIX standard is under revision. Because of the problems just listed, proposed text for the revised standard reverts to rules that correspond more closely to the original existing practice. The proposed rules have special cases that make it possible to produce a `\` preceding the matched text:

In a nutshell, at the runtime level, there are now three special sequences of characters (`\\\&`, `\\&`, and `\&`) whereas historically there was only one. However, as in the historical case, any `\` that is not part of one of these three sequences is not special and appears in the output literally.

gawk 3.0 and 3.1 follow these proposed POSIX rules for `sub` and `gsub`. Whether these proposed rules will actually become codified into the standard is unknown at this point. Subsequent *gawk* releases will track the standard and implement whatever the final version specifies; this book will be updated as well.†

The rules for `gensub` are considerably simpler. At the runtime level, whenever *gawk* sees a `\`, if the following character is a digit, then the text that matched the corresponding parenthesized subexpression is placed in the generated output. Otherwise, no matter what character follows the `\`, it appears in the generated text and the `\` does not, as shown in Table 8-3.

* This consequence was certainly unintended.

† As this book went to press, we learned that the POSIX standard will not use these rules. However, it was too late to change *gawk* for the 3.1 release. *gawk* behaves as described here.

Table 8-3. Escape Sequence Processing for gensub

You type	gensub sees	gensub generates
`&`	`&`	The matched text
`\\&`	`\&`	A literal `&`
`\\\\`	`\\`	A literal `\`
`\\\\&`	`\\&`	A literal `\`, then the matched text
`\\\\\\&`	`\\\&`	A literal `\&`
`\\q`	`\q`	A literal `q`

Because of the complexity of the lexical and runtime level processing and the special cases for `sub` and `gsub`, we recommend the use of *gawk* and `gensub` when you have to do substitutions.

Matching the Null String

In *awk*, the `*` operator can match the null string. This is particularly important for the `sub`, `gsub`, and `gensub` functions. For example:

```
$ echo abc | awk '{ gsub(/m*/, "X"); print }'
XaXbXcX
```

Although this makes a certain amount of sense, it can be surprising.

Input/Output Functions

The following functions relate to input/output (I/O). Optional parameters are enclosed in square brackets ([ ]):

`close(`*`filename`*`[, `*`how`*`])`

Close the file *filename* for input or output. Alternatively, the argument may be a shell command that was used for creating a coprocess, or for redirecting to or from a pipe; then the coprocess or pipe is closed. See the section "Closing Input and Output Redirections" in Chapter 4 for more information.

When closing a coprocess, it is occasionally useful to first close one end of the two-way pipe and then to close the other. This is done by providing a second argument to `close`. This second argument should be one of the two string values `"to"` or `"from"`, indicating which end of the pipe to close. Case in the string does not matter. See the section "Two-Way Communications with Another Process" in Chapter 10, which discusses this feature in more detail and gives an example.

`fflush([`*`filename`*`])`

Flush any buffered output associated with *filename*, which is either a file opened for writing or a shell command for redirecting output to a pipe or coprocess.

Many utility programs *buffer* their output; i.e., they save information to write to a disk file or terminal in memory until there is enough for it to be worthwhile to send the data to the output device. This is often more efficient than writing every little bit of information as soon as it is ready. However, sometimes it is necessary to force a program to *flush* its buffers; that is, write the information to its destination, even if a buffer is not full. This is the purpose of the `fflush` function—*gawk* also buffers its output and the `fflush` function forces *gawk* to flush its buffers.

`fflush` was added to the Bell Laboratories research version of *awk* in 1994; it is not part of the POSIX standard and is not available if *––posix* has been specified on the command line (see the section "Command-Line Options" in Chapter 11).

gawk extends the `fflush` function in two ways. The first is to allow no argument at all. In this case, the buffer for the standard output is flushed. The second is to allow the null string (`""`) as the argument. In this case, the buffers for *all* open output files and pipes are flushed.

`fflush` returns zero if the buffer is successfully flushed; otherwise, it returns −1. In the case where all buffers are flushed, the return value is zero only if all buffers were flushed successfully. Otherwise, it is −1, and *gawk* warns about the problem *filename*.

gawk also issues a warning message if you attempt to flush a file or pipe that was opened for reading (such as with `getline`), or if *filename* is not an open file, pipe, or coprocess. In such a case, `fflush` returns −1, as well.

`system(`*`command`*`)`

Executes operating-system commands and then return to the *awk* program. The `system` function executes the command given by the string *command*. It returns the status returned by the command that was executed as its value.

For example, if the following fragment of code is put in your *awk* program:

```
END {
    system("date | mail -s 'awk run done' root")
}
```

the system administrator is sent mail when the *awk* program finishes processing input and begins its end-of-input processing.

Note that redirecting `print` or `printf` into a pipe is often enough to accomplish your task. If you need to run many commands, it is more efficient to simply print them down a pipeline to the shell:

```
while (more stuff to do)
    print command | "/bin/sh"
close("/bin/sh")
```

However, if your *awk* program is interactive, `system` is useful for cranking up large self-contained programs, such as a shell or an editor. Some operating systems cannot implement the `system` function. `system` causes a fatal error if it is not supported.

Interactive Versus Noninteractive Buffering

As a side point, buffering issues can be even more confusing, depending upon whether your program is *interactive*, i.e., communicating with a user sitting at a keyboard.*

Interactive programs generally *line buffer* their output; i.e., they write out every line. Noninteractive programs wait until they have a full buffer, which may be many lines of output. Here is an example of the difference:

```
$ awk '{ print $1 + $2 }'
1 1
2
2 3
5
Ctrl-d
```

Each line of output is printed immediately. Compare that behavior with this example:

```
$ awk '{ print $1 + $2 }' | cat
1 1
2 3
Ctrl-d
2
5
```

Here, no output is printed until after the Ctrl-d is typed, because it is all buffered and sent down the pipe to *cat* in one shot.

* A program is interactive if the standard output is connected to a terminal device.

Controlling Output Buffering with system

The `fflush` function provides explicit control over output buffering for individual files and pipes. However, its use is not portable to many other *awk* implementations. An alternative method to flush output buffers is to call `system` with a null string as its argument:

```
system("")   # flush output
```

gawk treats this use of the `system` function as a special case and is smart enough not to run a shell (or other command interpreter) with the empty command. Therefore, with *gawk*, this idiom is not only useful, it is also efficient. While this method should work with other *awk* implementations, it does not necessarily avoid starting an unnecessary shell. (Other implementations may only flush the buffer associated with the standard output and not necessarily all buffered output.)

If you think about what a programmer expects, it makes sense that `system` should flush any pending output. The following program:

```
BEGIN {
    print "first print"
    system("echo system echo")
    print "second print"
}
```

must print:

```
first print
system echo
second print
```

and not:

```
system echo
first print
second print
```

If *awk* did not flush its buffers before calling `system`, you would see the latter (undesirable) output.

Using gawk's Timestamp Functions

awk programs are commonly used to process log files containing timestamp information, indicating when a particular log record was written. Many programs log their timestamp in the form returned by the `time` system call, which is the number of seconds since a particular epoch. On POSIX-compliant systems, it is the number of seconds since 1970-01-01 00:00:00 UTC, not counting leap seconds.* All known POSIX-compliant systems support timestamps from 0 through $2^{31} - 1$, which is

* See the Glossary, especially the entries "Epoch" and "UTC."

sufficient to represent times through 2038-01-19 03:14:07 UTC. Many systems support a wider range of timestamps, including negative timestamps that represent times before the epoch.

In order to make it easier to process such log files and to produce useful reports, *gawk* provides the following functions for working with timestamps. They are *gawk* extensions; they are not specified in the POSIX standard, nor are they in any other known version of *awk*.[*] Optional parameters are enclosed in square brackets ([ ]):

`systime()`
: This function returns the current time as the number of seconds since the system epoch. On POSIX systems, this is the number of seconds since 1970-01-01 00:00:00 UTC, not counting leap seconds. It may be a different number on other systems.

`mktime(`*`datespec`*`)`
: This function turns *datespec* into a timestamp in the same form as is returned by `systime`. It is similar to the function of the same name in ISO C. The argument, *datespec*, is a string of the form `"YYYY MM DD HH MM SS [DST]"`. The string consists of six or seven numbers representing, respectively, the full year including century, the month from 1 to 12, the day of the month from 1 to 31, the hour of the day from 0 to 23, the minute from 0 to 59, the second from 0 to 60,[†] and an optional daylight-savings flag.

  The values of these numbers need not be within the ranges specified; for example, an hour of −1 means 1 hour before midnight. The origin-zero Gregorian calendar is assumed, with year 0 preceding year 1 and year −1 preceding year 0. The time is assumed to be in the local timezone. If the daylight-savings flag is positive, the time is assumed to be daylight savings time; if zero, the time is assumed to be standard time; and if negative (the default), `mktime` attempts to determine whether daylight savings time is in effect for the specified time.

  If *datespec* does not contain enough elements or if the resulting time is out of range, `mktime` returns −1.

`strftime([`*`format`*` [, `*`timestamp`*`]])`
: This function returns a string. It is similar to the function of the same name in ISO C. The time specified by *timestamp* is used to produce a string, based on the contents of the *format* string. The *timestamp* is in the same format as the

[*] The GNU *date* utility can also do many of the things described here. Its use may be preferable for simple time-related operations in shell scripts.

[†] Occasionally there are minutes in a year with a leap second, which is why the seconds can go up to 60.

value returned by the `systime` function. If no *timestamp* argument is supplied, *gawk* uses the current time of day as the timestamp. If no *format* argument is supplied, `strftime` uses `"%a %b %d %H:%M:%S %Z %Y"`. This format string produces output that is (almost) equivalent to that of the *date* utility. (Versions of *gawk* prior to 3.0 require the *format* argument.)

The `systime` function allows you to compare a timestamp from a log file with the current time of day. In particular, it is easy to determine how long ago a particular record was logged. It also allows you to produce log records using the "seconds since the epoch" format.

The `mktime` function allows you to convert a textual representation of a date and time into a timestamp. This makes it easy to do before/after comparisons of dates and times, particularly when dealing with date and time data coming from an external source, such as a log file.

The `strftime` function allows you to easily turn a timestamp into human-readable information. It is similar in nature to the `sprintf` function (see the section "String-Manipulation Functions" earlier in this chapter), in that it copies nonformat specification characters verbatim to the returned string, while substituting date and time values for format specifications in the *format* string.

`strftime` is guaranteed by the 1999 ISO C standard* to support the following date format specifications:

`%a` The locale's abbreviated weekday name.

`%A` The locale's full weekday name.

`%b` The locale's abbreviated month name.

`%B` The locale's full month name.

`%c` The locale's "appropriate" date and time representation. (This is `%A %B %d %T %Y` in the `"C"` locale.)

`%C` The century. This is the year divided by 100 and truncated to the next lower integer.

`%d` The day of the month as a decimal number (01–31).

`%D` Equivalent to specifying `%m/%d/%y`.

`%e` The day of the month, padded with a space if it is only one digit.

* As this is a recent standard, not every system's `strftime` necessarily supports all of the conversions listed here.

`%F` Equivalent to specifying `%Y-%m-%d`. This is the ISO 8601 date format.

`%g` The year modulo 100 of the ISO week number, as a decimal number (00–99). For example, January 1, 1993 is in week 53 of 1992. Thus, the year of its ISO week number is 1992, even though its year is 1993. Similarly, December 31, 1973 is in week 1 of 1974. Thus, the year of its ISO week number is 1974, even though its year is 1973.

`%G` The full year of the ISO week number, as a decimal number.

`%h` Equivalent to `%b`.

`%H` The hour (24-hour clock) as a decimal number (00–23).

`%I` The hour (12-hour clock) as a decimal number (01–12).

`%j` The day of the year as a decimal number (001–366).

`%m` The month as a decimal number (01–12).

`%M` The minute as a decimal number (00–59).

`%n` A newline character (ASCII LF).

`%p` The locale's equivalent of the AM/PM designations associated with a 12-hour clock.

`%r` The locale's 12-hour clock time. (This is `%I:%M:%S %p` in the `"C"` locale.)

`%R` Equivalent to specifying `%H:%M`.

`%S` The second as a decimal number (00–60).

`%t` A tab character.

`%T` Equivalent to specifying `%H:%M:%S`.

`%u` The weekday as a decimal number (1–7). Monday is day one.

`%U` The week number of the year (the first Sunday as the first day of week one) as a decimal number (00–53).

`%V` The week number of the year (the first Monday as the first day of week one) as a decimal number (01–53). The method for determining the week number is as specified by ISO 8601. (To wit: if the week containing January 1 has four or more days in the new year, then it is week one; otherwise, it is week 53 of the previous year and the next week is week one.)

`%w` The weekday as a decimal number (0–6). Sunday is day zero.

`%W` The week number of the year (the first Monday as the first day of week one) as a decimal number (00–53).

`%x` The locale's "appropriate" date representation. (This is `%A %B %d %Y` in the `"C"` locale.)

`%X` The locale's "appropriate" time representation. (This is `%T` in the `"C"` locale.)

`%y` The year modulo 100 as a decimal number (00–99).

`%Y` The full year as a decimal number (e.g., 1995).

`%z` The time zone offset in a +HHMM format (e.g., the format necessary to produce RFC 822/RFC 1036 date headers).

`%Z` The time zone name or abbreviation; no characters if no time zone is determinable.

`%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH`
`%OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy`
"Alternate representations" for the specifications that use only the second letter (`%c`, `%C`, and so on).* (These facilitate compliance with the POSIX *date* utility.)

`%%` A literal `%`.

If a conversion specifier is not one of the above, the behavior is undefined.†

Informally, a *locale* is the geographic place in which a program is meant to run. For example, a common way to abbreviate the date September 4, 1991 in the United States is "9/4/91." In many countries in Europe, however, it is abbreviated "4.9.91." Thus, the `%x` specification in a `"US"` locale might produce `9/4/91`, while in a `"EUROPE"` locale, it might produce `4.9.91`. The ISO C standard defines a default `"C"` locale, which is an environment that is typical of what most C programmers are used to.

A public-domain C version of `strftime` is supplied with *gawk* for systems that are not yet fully standards-compliant. It supports all of the just listed format specifications. If that version is used to compile *gawk* (see Appendix B, *Installing gawk*), then the following additional format specifications are available:

`%k` The hour (24-hour clock) as a decimal number (0–23). Single-digit numbers are padded with a space.

`%l` The hour (12-hour clock) as a decimal number (1–12). Single-digit numbers are padded with a space.

* If you don't understand any of this, don't worry about it; these facilities are meant to make it easier to "internationalize" programs. Other internationalization features are described in Chapter 9, *Internationalization with gawk*.

† This is because ISO C leaves the behavior of the C version of `strftime` undefined and *gawk* uses the system's version of `strftime` if it's there. Typically, the conversion specifier either does not appear in the returned string or appears literally.

`%N` The “Emperor/Era” name. Equivalent to `%C`.

`%o` The “Emperor/Era” year. Equivalent to `%y`.

`%s` The time as a decimal timestamp in seconds since the epoch.

`%v` The date in VMS format (e.g., `20-JUN-1991`).

Additionally, the alternate representations are recognized but their normal representations are used.

This example is an *awk* implementation of the POSIX *date* utility. Normally, the *date* utility prints the current date and time of day in a well-known format. However, if you provide an argument to it that begins with a `+`, *date* copies nonformat specifier characters to the standard output and interprets the current time according to the format specifiers in the string. For example:

```
$ date '+Today is %A, %B %d, %Y.'
Today is Thursday, September 14, 2000.
```

Here is the *gawk* version of the *date* utility. It has a shell “wrapper” to handle the *–u* option, which requires that *date* run as if the time zone is set to UTC:

```
#! /bin/sh
#
# date --- approximate the P1003.2 'date' command

case $1 in
-u) TZ=UTC0     # use UTC
    export TZ
    shift ;;
esac

gawk 'BEGIN  {
    format = "%a %b %d %H:%M:%S %Z %Y"
    exitval = 0

    if (ARGC > 2)
        exitval = 1
    else if (ARGC == 2) {
        format = ARGV[1]
        if (format ~ /^\+/)
            format = substr(format, 2)   # remove leading +
    }
    print strftime(format)
    exit exitval
}' "$@"
```

Bit-Manipulation Functions of gawk

Many languages provide the ability to perform *bitwise* operations on two integer numbers. In other words, the operation is performed on each successive pair of bits in the operands. Three common operations are bitwise AND, OR, and XOR. The operations are described in Table 8-4.

Table 8-4. Bitwise Operations

	Bit Operator					
	AND		OR		XOR	
Operands	0	1	0	1	0	1
0	0	0	0	1	0	1
1	0	1	1	1	1	0

As you can see, the result of an AND operation is 1 only when *both* bits are 1. The result of an OR operation is 1 if *either* bit is 1. The result of an XOR operation is 1 if either bit is 1, but not both. The next operation is the *complement*; the complement of 1 is 0 and the complement of 0 is 1. Thus, this operation "flips" all the bits of a given value.

Finally, two other common operations are to shift the bits left or right. For example, if you have a bit string `10111001` and you shift it right by three bits, you end up with `00010111`.* If you start over again with `10111001` and shift it left by three bits, you end up with `11001000`. *gawk* provides built-in functions that implement the bitwise operations just described. They are:

`and(`*`v1, v2`*`)`
: Returns the bitwise AND of the values provided by *v1* and *v2*.

`or(`*`v1, v2`*`)`
: Returns the bitwise OR of the values provided by *v1* and *v2*.

`xor(`*`v1, v2`*`)`
: Returns the bitwise XOR of the values provided by *v1* and *v2*.

`compl(`*`val`*`)`
: Returns the bitwise complement of *val*.

* This example shows that 0's come in on the left side. For *gawk*, this is always true, but in some languages, it's possible to have the left side fill with 1's. Caveat emptor.

`lshift(`*`val`*`, `*`count`*`)`

Returns the value of *val*, shifted left by *count* bits.

`rshift(`*`val, count`*`)`

Returns the value of *val*, shifted right by *count* bits.

For all of these functions, first the double-precision floating-point value is converted to a C `unsigned long`, then the bitwise operation is performed and then the result is converted back into a C `double`. (If you don't understand this paragraph, don't worry about it.)

Here is a user-defined function (see the section "User-Defined Functions" later in this chapter) that illustrates the use of these functions:

```
# bits2str --- turn a byte into readable 1's and 0's

function bits2str(bits,        data, mask)
{
    if (bits == 0)
        return "0"

    mask = 1
    for (; bits != 0; bits = rshift(bits, 1))
        data = (and(bits, mask) ? "1" : "0") data

    while ((length(data) % 8) != 0)
        data = "0" data

    return data
}

BEGIN {
    printf "123 = %s\n", bits2str(123)
    printf "0123 = %s\n", bits2str(0123)
    printf "0x99 = %s\n", bits2str(0x99)
    comp = compl(0x99)
    printf "compl(0x99) = %#x = %s\n", comp, bits2str(comp)
    shift = lshift(0x99, 2)
    printf "lshift(0x99, 2) = %#x = %s\n", shift, bits2str(shift)
    shift = rshift(0x99, 2)
    printf "rshift(0x99, 2) = %#x = %s\n", shift, bits2str(shift)
}
```

This program produces the following output when run:

```
$ gawk -f testbits.awk
123 = 01111011
0123 = 01010011
0x99 = 10011001
compl(0x99) = 0xffffff66 = 11111111111111111111111101100110
lshift(0x99, 2) = 0x264 = 0000001001100100
rshift(0x99, 2) = 0x26 = 00100110
```

The `bits2str` function turns a binary number into a string. The number `1` represents a binary value where the rightmost bit is set to 1. Using this mask, the function repeatedly checks the rightmost bit. ANDing the mask with the value indicates whether the rightmost bit is 1 or not. If so, a `"1"` is concatenated onto the front of the string. Otherwise, a `"0"` is added. The value is then shifted right by one bit and the loop continues until there are no more 1 bits.

If the initial value is zero it returns a simple `"0"`. Otherwise, at the end, it pads the value with zeros to represent multiples of 8-bit quantities. This is typical in modern computers.

The main code in the `BEGIN` rule shows the difference between the decimal and octal values for the same numbers (see the section "Octal and Hexadecimal Numbers" in Chapter 5, *Expressions*), and then demonstrates the results of the `compl`, `lshift`, and `rshift` functions.

Using gawk's String-Translation Functions

gawk provides facilities for internationalizing *awk* programs. These include the functions described in the following list. The descriptions here are purposely brief. See Chapter 9 for the full story. Optional parameters are enclosed in square brackets ([ ]):

`dcgettext(`*`string`* [`,` *`domain`* [`,` *`category`*]]`)`

This function returns the translation of *string* in text domain *domain* for locale category *category*. The default value for *domain* is the current value of `TEXTDOMAIN`. The default value for *category* is `"LC_MESSAGES"`.

`bindtextdomain(`*`directory`* [`,` *`domain`*]`)`

This function allows you to specify the directory in which *gawk* will look for message translation files, in case they will not or cannot be placed in the "standard" locations (e.g., during testing). It returns the directory in which *domain* is "bound."

The default *domain* is the value of `TEXTDOMAIN`. If *directory* is the null string (`""`), then `bindtextdomain` returns the current binding for the given *domain*.

User-Defined Functions

Complicated *awk* programs can often be simplified by defining your own functions. User-defined functions can be called just like built-in ones (see the section "Function Calls" in Chapter 5), but it is up to you to define them, i.e., to tell *awk* what they should do.

Function Definition Syntax

Definitions of functions can appear anywhere between the rules of an *awk* program. Thus, the general form of an *awk* program is extended to include sequences of rules *and* user-defined function definitions. There is no need to put the definition of a function before all uses of the function. This is because *awk* reads the entire program before starting to execute any of it.

The definition of a function named *name* looks like this:

```
function name(parameter-list)
{
    body-of-function
}
```

name is the name of the function to define. A valid function name is like a valid variable name: a sequence of letters, digits, and underscores that doesn't start with a digit. Within a single *awk* program, any particular name can only be used as a variable, array, or function.

parameter-list is a list of the function's arguments and local variable names, separated by commas. When the function is called, the argument names are used to hold the argument values given in the call. The local variables are initialized to the empty string. A function cannot have two parameters with the same name, nor may it have a parameter with the same name as the function itself.

The *body-of-function* consists of *awk* statements. It is the most important part of the definition, because it says what the function should actually *do*. The argument names exist to give the body a way to talk about the arguments; local variables exist to give the body places to keep temporary values.

Argument names are not distinguished syntactically from local variable names. Instead, the number of arguments supplied when the function is called determines how many argument variables there are. Thus, if three argument values are given, the first three names in *parameter-list* are arguments and the rest are local variables.

It follows that if the number of arguments is not the same in all calls to the function, some of the names in *parameter-list* may be arguments on some occasions and local variables on others. Another way to think of this is that omitted arguments default to the null string.

Usually when you write a function, you know how many names you intend to use for arguments and how many you intend to use as local variables. It is conventional to place some extra space between the arguments and the local variables, in order to document how your function is supposed to be used.

During execution of the function body, the arguments and local variable values hide, or *shadow*, any variables of the same names used in the rest of the program. The shadowed variables are not accessible in the function definition, because there is no way to name them while their names have been taken away for the local variables. All other variables used in the *awk* program can be referenced or set normally in the function's body.

The arguments and local variables last only as long as the function body is executing. Once the body finishes, you can once again access the variables that were shadowed while the function was running.

The function body can contain expressions that call functions. They can even call this function, either directly or by way of another function. When this happens, we say the function is *recursive*. The act of a function calling itself is called *recursion*.

In many *awk* implementations, including *gawk*, the keyword `function` may be abbreviated `func`. However, POSIX only specifies the use of the keyword `function`. This actually has some practical implications. If *gawk* is in POSIX-compatibility mode (see the section "Command-Line Options" in Chapter 11), then the following statement does *not* define a function:

```
func foo() { a = sqrt($1) ; print a }
```

Instead it defines a rule that, for each record, concatenates the value of the variable `func` with the return value of the function `foo`. If the resulting string is non-null, the action is executed. This is probably not what is desired. (*awk* accepts this input as syntactically valid, because functions may be used before they are defined in *awk* programs.)

To ensure that your *awk* programs are portable, always use the keyword `function` when defining a function.

Function Definition Examples

Here is an example of a user-defined function, called `myprint`, that takes a number and prints it in a specific format:

```
function myprint(num)
{
    printf "%6.3g\n", num
}
```

To illustrate, here is an *awk* rule that uses our `myprint` function:

```
$3 > 0     { myprint($3) }
```

This program prints, in our special format, all the third fields that contain a positive number in our input. Therefore, when given the following:

```
 1.2   3.4    5.6   7.8
 9.10 11.12 -13.14 15.16
17.18 19.20  21.22 23.24
```

this program, using our function to format the results, prints:

```
 5.6
21.2
```

This function deletes all the elements in an array:

```
function delarray(a,    i)
{
    for (i in a)
        delete a[i]
}
```

When working with arrays, it is often necessary to delete all the elements in an array and start over with a new list of elements (see the section "The delete Statement" in Chapter 7). Instead of having to repeat this loop everywhere that you need to clear out an array, your program can just call `delarray`. (This guarantees portability. The use of `delete` *`array`* to delete the contents of an entire array is a nonstandard extension.)

The following is an example of a recursive function. It takes a string as an input parameter and returns the string in backwards order. Recursive functions must always have a test that stops the recursion. In this case, the recursion terminates when the starting position is zero, i.e., when there are no more characters left in the string:

```
function rev(str, start)
{
    if (start == 0)
        return ""

    return (substr(str, start, 1) rev(str, start - 1))
}
```

If this function is in a file named *rev.awk*, it can be tested this way:

```
$ echo "Don't Panic!" |
> gawk --source '{ print rev($0, length($0)) }' -f rev.awk
!cinaP t'noD
```

The C `ctime` function takes a timestamp and returns it in a string, formatted in a well-known fashion. The following example uses the built-in `strftime` function (see the section "Using gawk's Timestamp Functions" earlier in this chapter) to create an *awk* version of `ctime`:

```
# ctime.awk
#
# awk version of C ctime(3) function

function ctime(ts,    format)
{
    format = "%a %b %d %H:%M:%S %Z %Y"
    if (ts == 0)
        ts = systime()       # use current time as default
    return strftime(format, ts)
}
```

Calling User-Defined Functions

Calling a function means causing the function to run and do its job. A function call is an expression and its value is the value returned by the function.

A function call consists of the function name followed by the arguments in parentheses. *awk* expressions are what you write in the call for the arguments. Each time the call is executed, these expressions are evaluated, and the values are the actual arguments. For example, here is a call to `foo` with three arguments (the first being a string concatenation):

```
foo(x y, "lose", 4 * z)
```

Whitespace characters (spaces and tabs) are not allowed between the function name and the open-parenthesis of the argument list. If you write whitespace by mistake, *awk* might think that you mean to concatenate a variable with an expression in parentheses. However, it notices that you used a function name and not a variable name, and reports an error.

When a function is called, it is given a *copy* of the values of its arguments. This is known as *call by value*. The caller may use a variable as the expression for the argument, but the called function does not know this—it only knows what value the argument had. For example, if you write the following code:

```
foo = "bar"
z = myfunc(foo)
```

then you should not think of the argument to `myfunc` as being "the variable `foo`." Instead, think of the argument as the string value `"bar"`. If the function `myfunc` alters the values of its local variables, this has no effect on any other variables. Thus, if `myfunc` does this:

```
function myfunc(str)
{
    print str
    str = "zzz"
    print str
}
```

to change its first argument variable `str`, it does *not* change the value of `foo` in the caller. The role of `foo` in calling `myfunc` ended when its value (`"bar"`) was computed. If `str` also exists outside of `myfunc`, the function body cannot alter this outer value, because it is shadowed during the execution of `myfunc` and cannot be seen or changed from there.

However, when arrays are the parameters to functions, they are *not* copied. Instead, the array itself is made available for direct manipulation by the function. This is usually called *call by reference*. Changes made to an array parameter inside the body of a function *are* visible outside that function.

Changing an array parameter inside a function can be very dangerous if you do not watch what you are doing. For example:

```
function changeit(array, ind, nvalue)
{
    array[ind] = nvalue
}

BEGIN {
    a[1] = 1; a[2] = 2; a[3] = 3
    changeit(a, 2, "two")
    printf "a[1] = %s, a[2] = %s, a[3] = %s\n",
            a[1], a[2], a[3]
}
```

prints `a[1] = 1, a[2] = two, a[3] = 3`, because `changeit` stores `"two"` in the second element of `a`.

Some *awk* implementations allow you to call a function that has not been defined. They only report a problem at runtime when the program actually tries to call the function. For example:

```
BEGIN {
    if (0)
        foo()
    else
        bar()
}
function bar() { ... }
# note that 'foo' is not defined
```

Because the `if` statement will never be true, it is not really a problem that `foo` has not been defined. Usually, though, it is a problem if a program calls an undefined function.

If *--lint* is specified (see the section "Command-Line Options" in Chapter 11), *gawk* reports calls to undefined functions.

Some *awk* implementations generate a runtime error if you use the `next` statement (see the section "The next Statement" in Chapter 6, *Patterns, Actions, and Variables*) inside a user-defined function. *gawk* does not have this limitation.

The return Statement

The body of a user-defined function can contain a `return` statement. This statement returns control to the calling part of the *awk* program. It can also be used to return a value for use in the rest of the *awk* program. It looks like this:

```
return [expression]
```

The *expression* part is optional. If it is omitted, then the returned value is undefined, and therefore, unpredictable.

A `return` statement with no value expression is assumed at the end of every function definition. So if control reaches the end of the function body, then the function returns an unpredictable value. *awk* does *not* warn you if you use the return value of such a function.

Sometimes, you want to write a function for what it does, not for what it returns. Such a function corresponds to a `void` function in C or to a `procedure` in Pascal. Thus, it may be appropriate to not return any value; simply bear in mind that if you use the return value of such a function, you do so at your own risk.

The following is an example of a user-defined function that returns a value for the largest number among the elements of an array:

```
function maxelt(vec,   i, ret)
{
    for (i in vec) {
        if (ret == "" || vec[i] > ret)
            ret = vec[i]
    }
    return ret
}
```

You call `maxelt` with one argument, which is an array name. The local variables `i` and `ret` are not intended to be arguments; while there is nothing to stop you from passing several arguments to `maxelt`, the results would be strange. The extra space before `i` in the function parameter list indicates that `i` and `ret` are not supposd to be arguments. You should follow this convention when defining functions.

The following program uses the `maxelt` function. It loads an array, calls `maxelt`, and then reports the maximum number in that array:

```
function maxelt(vec,   i, ret)
{
    for (i in vec) {
        if (ret == "" || vec[i] > ret)
            ret = vec[i]
    }
    return ret
}

# Load all fields of each record into nums.
{
    for(i = 1; i <= NF; i++)
        nums[NR, i] = $i
}

END {
    print maxelt(nums)
}
```

Given the following input:

```
 1 5 23 8 16
44 3 5 2 8 26
256 291 1396 2962 100
-6 467 998 1101
99385 11 0 225
```

the program reports (predictably) that `99385` is the largest number in the array.

Functions and Their Effects on Variable Typing

awk is a very fluid language. It is possible that *awk* can't tell if an identifier represents a regular variable or an array until runtime. Here is an annotated sample program:

```
function foo(a)
{
    a[1] = 1   # parameter is an array
}

BEGIN {
    b = 1
    foo(b)  # invalid: fatal type mismatch

    foo(x)  # x uninitialized, becomes an array dynamically
    x = 1   # now not allowed, runtime error
}
```

Usually, such things aren't a big issue, but it's worth being aware of them.