5

Expressions

In this chapter:

Expressions are the basic building blocks of *awk* patterns and actions. An expression evaluates to a value that you can print, test, or pass to a function. Additionally, an expression can assign a new value to a variable or a field by using an assignment operator.

An expression can serve as a pattern or action statement on its own. Most other kinds of statements contain one or more expressions that specify the data on which to operate. As in other languages, expressions in *awk* include variables, array references, constants, and function calls, as well as combinations of these with various operators.

9 October 2001 01:41

Constant Expressions

The simplest type of expression is the *constant*, which always has the same value. There are three types of constants: numeric, string, and regular expression.

Each is used in the appropriate context when you need a data value that isn't going to change. Numeric constants can have different forms, but are stored identically internally.

Numeric and String Constants

A *numeric constant* stands for a number. This number can be an integer, a decimal fraction, or a number in scientific (exponential) notation.* Here are some examples of numeric constants that all have the same value:

```
105
1.05e+2
1050e-1
```

A string constant consists of a sequence of characters enclosed in double-quotation marks. For example:

```
"parrot"
```

represents the string whose contents are `parrot`. Strings in *gawk* can be of any length, and they can contain any of the possible eight-bit ASCII characters including ASCII NUL (character code zero). Other *awk* implementations may have difficulty with some character codes.

Octal and Hexadecimal Numbers

In *awk*, all numbers are in decimal; i.e., base 10. Many other programming languages allow you to specify numbers in other bases, often octal (base 8) and hexadecimal (base 16). In octal, the numbers go 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, etc. Just as `11`, in decimal, is 1 times 10 plus 1, so `11`, in octal, is 1 times 8 plus 1. This equals 9 in decimal. In hexadecimal, there are 16 digits. Since the everyday decimal number system only has 10 digits (`0`–`9`), the letters `a` through `f` are used to represent the rest. (Case in the letters is usually irrelevant; hexadecimal `a` and `A` have the same value.) Thus, `11`, in hexadecimal, is 1 times 16 plus 1, which equals 17 in decimal.

Just by looking at plain `11`, you can't tell what base it's in. So, in C, C++, and other languages derived from C, there is a special notation to help signify the base.

* The internal representation of all numbers, including integers, uses double-precision floating-point numbers. On most modern systems, these are in IEEE 754 standard format.

Octal numbers start with a leading `0`, and hexadecimal numbers start with a leading `0x` or `0X`:

`11`	Decimal value 11.
`011`	Octal 11, decimal value 9.
`0x11`	Hexadecimal 11, decimal value 17.

This example shows the difference:

```
$ gawk 'BEGIN { printf "%d, %d, %d\n", 011, 11, 0x11 }'
9, 11, 17
```

Being able to use octal and hexadecimal constants in your programs is most useful when working with data that cannot be represented conveniently as characters or as regular numbers, such as binary data of various sorts.

gawk allows the use of octal and hexadecimal constants in your program text. However, such numbers in the input data are not treated differently; doing so by default would break old programs. (If you really need to do this, use the *––non–decimal–data* command-line option; see the section "Allowing Nondecimal Input Data" in Chapter 10, *Advanced Features of gawk*.) If you have octal or hexadecimal data, you can use the `strtonum` function (see the section "String-Manipulation Functions" in Chapter 8, *Functions*) to convert the data into a number. Most of the time, you will want to use octal or hexadecimal constants when working with the built-in bit manipulation functions; see the section "Bit-Manipulation Functions of gawk" in Chapter 8 for more information.

Unlike some early C implementations, `8` and `9` are not valid in octal constants; e.g., *gawk* treats `018` as decimal 18:

```
$ gawk 'BEGIN { print "021 is", 021 ; print 018 }'
021 is 17
18
```

Octal and hexadecimal source code constants are a *gawk* extension. If *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*), they are not available.

Regular Expression Constants

A regexp constant is a regular expression description enclosed in slashes, such as `/^beginning and end$/`. Most regexps used in *awk* programs are constant, but the `~` and `!~` matching operators can also match computed or "dynamic" regexps (which are just ordinary strings or variables that contain a regexp).

A Constant's Base Does Not Affect Its Value

Once a numeric constant has been converted internally into a number, *gawk* no longer remembers what the original form of the constant was; the internal value is always used. This has particular consequences for conversion of numbers to strings:

```
$ gawk 'BEGIN { printf "0x11 is <%s>\n", 0x11 }'
0x11 is <17>
```

Using Regular Expression Constants

When used on the righthand side of the `~` or `!~` operators, a regexp constant merely stands for the regexp that is to be matched. However, regexp constants (such as `/foo/`) may be used like simple expressions. When a regexp constant appears by itself, it has the same meaning as if it appeared in a pattern, i.e., `($0 ~ /foo/)`. (d.c.) See the section "Expressions as Patterns" in Chapter 6, *Patterns, Actions, and Variables*. This means that the following two code segments:

```
if ($0 ~ /barfly/ || $0 ~ /camelot/)
    print "found"
```

and:

```
if (/barfly/ || /camelot/)
    print "found"
```

are exactly equivalent. One rather bizarre consequence of this rule is that the following Boolean expression is valid, but does not do what the user probably intended:

```
# note that /foo/ is on the left of the ~
if (/foo/ ~ $1) print "found foo"
```

This code is "obviously" testing `$1` for a match against the regexp `/foo/`. But in fact, the expression `/foo/ ~ $1` actually means `($0 ~ /foo/) ~ $1`. In other words, first match the input record against the regexp `/foo/`. The result is either zero or one, depending upon the success or failure of the match. That result is then matched against the first field in the record. Because it is unlikely that you would ever really want to make this kind of test, *gawk* issues a warning when it sees this construct in a program. Another consequence of this rule is that the assignment statement:

```
matches = /foo/
```

assigns either zero or one to the variable `matches`, depending upon the contents of the current input record. This feature of the language has never been well documented until the POSIX specification.

Constant regular expressions are also used as the first argument for the `gensub`, `sub`, and `gsub` functions, and as the second argument of the `match` function (see the section "String-Manipulation Functions" in Chapter 8). Modern implementations of *awk*, including *gawk*, allow the third argument of `split` to be a regexp constant, but some older implementations do not. (d.c.) This can lead to confusion when attempting to use regexp constants as arguments to user-defined functions (see the section "User-Defined Functions" in Chapter 8). For example:

```
function mysub(pat, repl, str, global)
{
    if (global)
        gsub(pat, repl, str)
    else
        sub(pat, repl, str)
    return str
}

{
    ...
    text = "hi! hi yourself!"
    mysub(/hi/, "howdy", text, 1)
    ...
}
```

In this example, the programmer wants to pass a regexp constant to the user-defined function `mysub`, which in turn passes it on to either `sub` or `gsub`. However, what really happens is that the `pat` parameter is either one or zero, depending upon whether or not `$0` matches `/hi/`. *gawk* issues a warning when it sees a regexp constant used as a parameter to a user-defined function, since passing a truth value in this way is probably not what was intended.

Variables

Variables are ways of storing values at one point in your program for use later in another part of your program. They can be manipulated entirely within the program text, and they can also be assigned values on the *awk* command line.

Using Variables in a Program

Variables let you give names to values and refer to them later. Variables have already been used in many of the examples. The name of a variable must be a sequence of letters, digits, or underscores, and it may not begin with a digit. Case is significant in variable names; `a` and `A` are distinct variables.

A variable name is a valid expression by itself; it represents the variable's current value. Variables are given new values with *assignment operators*, *increment operators*, and *decrement operators*. See the section "Assignment Expressions" later in this chapter.

A few variables have special built-in meanings, such as `FS` (the field separator), and `NF` (the number of fields in the current input record). See the section "Built-in Variables" in Chapter 6 for a list of the built-in variables. These built-in variables can be used and assigned just like all other variables, but their values are also used or changed automatically by *awk*. All built-in variables' names are entirely uppercase.

Variables in *awk* can be assigned either numeric or string values. The kind of value a variable holds can change over the life of a program. By default, variables are initialized to the empty string, which is zero if converted to a number. There is no need to "initialize" each variable explicitly in *awk*, which is what you would do in C and in most other traditional languages.

Assigning Variables on the Command Line

Any *awk* variable can be set by including a *variable assignment* among the arguments on the command line when *awk* is invoked (see the section "Other Command-Line Arguments" in Chapter 11). Such an assignment has the following form:

```
variable=text
```

With it, a variable is set either at the beginning of the *awk* run or in between input files. When the assignment is preceded with the *–v* option, as in the following:

```
-v variable=text
```

the variable is set at the very beginning, even before the `BEGIN` rules are run. The *–v* option and its assignment must precede all the filename arguments, as well as the program text. (See the section "Command-Line Options" in Chapter 11 for more information about the *–v* option.) Otherwise, the variable assignment is performed at a time determined by its position among the input file arguments—after the processing of the preceding input file argument. For example:

```
awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
```

prints the value of field number `n` for all input records. Before the first file is read, the command line sets the variable `n` equal to four. This causes the fourth field to be printed in lines from the file *inventory-shipped*. After the first file has finished, but before the second file is started, `n` is set to two, so that the second field is printed in lines from *BBS-list*:

```
$ awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
15
24
...
555-5553
555-3412
...
```

Command-line arguments are made available for explicit examination by the *awk* program in the `ARGV` array (see the section "Using ARGC and ARGV" in Chapter 6). *awk* processes the values of command-line assignments for escape sequences (see the section "Escape Sequences" in Chapter 2, *Regular Expressions*). (d.c.)

Conversion of Strings and Numbers

Strings are converted to numbers and numbers are converted to strings, if the context of the *awk* program demands it. For example, if the value of either `foo` or `bar` in the expression `foo + bar` happens to be a string, it is converted to a number before the addition is performed. If numeric values appear in string concatenation, they are converted to strings. Consider the following:

```
two = 2; three = 3
print (two three) + 4
```

This prints the (numeric) value 27. The numeric values of the variables `two` and `three` are converted to strings and concatenated together. The resulting string is converted back to the number 23, to which 4 is then added.

If, for some reason, you need to force a number to be converted to a string, concatenate the empty string, `""`, with that number. To force a string to be converted to a number, add zero to that string. A string is converted to a number by interpreting any numeric prefix of the string as numerals: `"2.5"` converts to 2.5, `"1e3"` converts to 1000, and `"25fix"` has a numeric value of 25. Strings that can't be interpreted as valid numbers convert to zero.

The exact manner in which numbers are converted into strings is controlled by the *awk* built-in variable `CONVFMT` (see the section "Built-in Variables" in Chapter 6). Numbers are converted using the `sprintf` function with `CONVFMT` as the format specifier (see the section "String-Manipulation Functions" in Chapter 8).

`CONVFMT`'s default value is `"%.6g"`, which prints a value with at least six significant digits. For some applications, you might want to change it to specify more precision. On most modern machines, 17 digits is enough to capture a floating-point number's value exactly, most of the time.*

* Pathological cases can require up to 752 digits (!), but we doubt that you need to worry about this.

Strange results can occur if you set CONVFMT to a string that doesn't tell sprintf how to format floating-point numbers in a useful way. For example, if you forget the % in the format, *awk* converts all numbers to the same constant string. As a special case, if a number is an integer, then the result of converting it to a string is *always* an integer, no matter what the value of CONVFMT may be. Given the following code fragment:

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

b has the value "12", not "12.00". (d.c.)

Prior to the POSIX standard, *awk* used the value of OFMT for converting numbers to strings. OFMT specifies the output format to use when printing numbers with print. CONVFMT was introduced in order to separate the semantics of conversion from the semantics of printing. Both CONVFMT and OFMT have the same default value: "%.6g". In the vast majority of cases, old *awk* programs do not change their behavior. However, these semantics for OFMT are something to keep in mind if you must port your new style program to older implementations of *awk*. We recommend that instead of changing your programs, just port *gawk* itself. See the section "The print Statement" in Chapter 4, *Printing Output*, for more information on the print statement.

Arithmetic Operators

The *awk* language uses the common arithmetic operators when evaluating expressions. All of these arithmetic operators follow normal precedence rules and work as you would expect them to.

The following example uses a file named *grades*, which contains a list of student names as well as three test scores per student (it's a small class):

```
Pat   100 97 58
Sandy  84 72 93
Chris  72 92 89
```

This programs takes the file *grades* and prints the average of the scores:

```
$ awk '{ sum = $2 + $3 + $4 ; avg = sum / 3
>        print $1, avg }' grades
Pat 85
Sandy 83
Chris 84.3333
```

The following list provides the arithmetic operators in *awk*, in order from the highest precedence to the lowest:

`- x` Negation.

`+ x` Unary plus; the expression is converted to a number.

`x ^ y`
`x ** y`
: Exponentiation; *x* raised to the *y* power. `2 ^ 3` has the value eight; the character sequence `**` is equivalent to `^`.

`x * y`
: Multiplication.

`x / y`
: Division; because all numbers in *awk* are floating-point numbers, the result is *not* rounded to an integer—`3 / 4` has the value 0.75. (It is a common mistake, especially for C programmers, to forget that *all* numbers in *awk* are floating-point, and that division of integer-looking constants produces a real number, not an integer.)

`x % y`
: Remainder; further discussion is provided in the text, just after this list.

`x + y`
: Addition.

`x - y`
: Subtraction.

Unary plus and minus have the same precedence, the multiplication operators all have the same precedence, and addition and subtraction have the same precedence.

When computing the remainder of `x % y`, the quotient is rounded toward zero to an integer and multiplied by *y*. This result is subtracted from *x*; this operation is sometimes known as "trunc-mod." The following relation always holds:

```
b * int(a / b) + (a % b) == a
```

One possibly undesirable effect of this definition of remainder is that `x % y` is negative if *x* is negative. Thus:

```
-17 % 8 = -1
```

In other *awk* implementations, the signedness of the remainder may be machine-dependent.

The POSIX standard only specifies the use of ^ for exponentiation. For maximum portability, do not use the ** operator.

String Concatenation

There is only one string operation: concatenation. It does not have a specific operator to represent it. Instead, concatenation is performed by writing expressions next to one another, with no operator. For example:

```
$ awk '{ print "Field number one: " $1 }' BBS-list
Field number one: aardvark
Field number one: alpo-net
...
```

Without the space in the string constant after the `:`, the line runs together. For example:

```
$ awk '{ print "Field number one:" $1 }' BBS-list
Field number one:aardvark
Field number one:alpo-net
...
```

Because string concatenation does not have an explicit operator, it is often necessary to insure that it happens at the right time by using parentheses to enclose the items to concatenate. For example, the following code fragment does not concatenate `file` and `name` as you might expect:

```
file = "file"
name = "name"
print "something meaningful" > file name
```

It is necessary to use the following:

```
print "something meaningful" > (file name)
```

Parentheses should be used around concatenation in all but the most common contexts, such as on the righthand side of `=`. Be careful about the kinds of expressions used in string concatenation. In particular, the order of evaluation of expressions used for concatenation is undefined in the *awk* language. Consider this example:

```
BEGIN {
    a = "don't"
    print (a " " (a = "panic"))
}
```

It is not defined whether the assignment to `a` happens before or after the value of `a` is retrieved for producing the concatenated value. The result could be either `don't panic`, or `panic panic`. The precedence of concatenation, when mixed with other operators, is often counter-intuitive. Consider this example:

```
$ awk 'BEGIN { print -12 " " -24 }'
-12-24
```

This "obviously" is concatenating −12, a space, and −24. But where did the space disappear to? The answer lies in the combination of operator precedences and *awk*'s automatic conversion rules. To get the desired result, write the program in the following manner:

```
$ awk 'BEGIN { print -12 " " (-24) }'
-12 -24
```

This forces *awk* to treat the - on the `-24` as unary. Otherwise, it's parsed as follows:

```
   -12 (" " - 24)
→ -12 (0 - 24)
→ -12 (-24)
→ -12-24
```

As mentioned earlier, when doing concatenation, *parenthesize*. Otherwise, you're never quite sure what you'll get.

Assignment Expressions

An *assignment* is an expression that stores a (usually different) value into a variable. For example, let's assign the value one to the variable `z`:

```
z = 1
```

After this expression is executed, the variable `z` has the value one. Whatever old value `z` had before the assignment is forgotten.

Assignments can also store string values. For example, the following stores the value `"this food is good"` in the variable `message`:

```
thing = "food"
predicate = "good"
message = "this " thing " is " predicate
```

This also illustrates string concatenation. The = sign is called an *assignment operator*. It is the simplest assignment operator because the value of the righthand operand is stored unchanged. Most operators (addition, concatenation, and so on) have no effect except to compute a value. If the value isn't used, there's no reason

to use the operator. An assignment operator is different; it does produce a value, but even if you ignore it, the assignment still makes itself felt through the alteration of the variable. We call this a *side effect*.

The lefthand operand of an assignment need not be a variable (see the section "Variables" earlier in this chapter); it can also be a field (see the section "Changing the Contents of a Field in Chapter 3, *Reading Input Files*) or an array element (see Chapter 7, *Arrays in awk*). These are all called *lvalues*, which means they can appear on the lefthand side of an assignment operator. The righthand operand may be any expression; it produces the new value that the assignment stores in the specified variable, field, or array element. (Such values are called *rvalues*.)

It is important to note that variables do *not* have permanent types. A variable's type is simply the type of whatever value it happens to hold at the moment. In the following program fragment, the variable `foo` has a numeric value at first, and a string value later on:

```
foo = 1
print foo
foo = "bar"
print foo
```

When the second assignment gives `foo` a string value, the fact that it previously had a numeric value is forgotten.

String values that do not begin with a digit have a numeric value of zero. After executing the following code, the value of `foo` is five:

```
foo = "a string"
foo = foo + 5
```

Using a variable as a number and then later as a string can be confusing and is poor programming style. The previous two examples illustrate how *awk* works, *not* how you should write *your* programs!

An assignment is an expression, so it has a value—the same value that is assigned. Thus, `z = 1` is an expression with the value one. One consequence of this is that you can write multiple assignments together, such as:

```
x = y = z = 5
```

This example stores the value five in all three variables (`x`, `y`, and `z`). It does so because the value of `z = 5`, which is five, is stored into `y` and then the value of `y = z = 5`, which is five, is stored into `x`.

Assignments may be used anywhere an expression is called for. For example, it is valid to write `x != (y = 1)` to set `y` to one, and then test whether `x` equals one. But this style tends to make programs hard to read; such nesting of assignments should be avoided, except perhaps in a one-shot program.

Aside from `=`, there are several other assignment operators that do arithmetic with the old value of the variable. For example, the operator `+=` computes a new value by adding the righthand value to the old value of the variable. Thus, the following assignment adds five to the value of `foo`:

```
foo += 5
```

This is equivalent to the following:

```
foo = foo + 5
```

Use whichever makes the meaning of your program clearer.

There are situations where using `+=` (or any assignment operator) is *not* the same as simply repeating the lefthand operand in the righthand expression. For example:

```
# Thanks to Pat Rankin for this example
BEGIN  {
    foo[rand()] += 5
    for (x in foo)
        print x, foo[x]

    bar[rand()] = bar[rand()] + 5
    for (x in bar)
        print x, bar[x]
}
```

The indices of `bar` are practically guaranteed to be different, because `rand` returns different values each time it is called. (Arrays and the `rand` function haven't been covered yet. See Chapter 7 and the section "Numeric Functions" in Chapter 8 for more information.) This example illustrates an important fact about assignment operators: the lefthand expression is only evaluated *once*. It is up to the implementation as to which expression is evaluated first, the lefthand or the righthand. Consider this example:

```
i = 1
a[i += 2] = i + 1
```

The value of `a[3]` could be either two or four.

Table 5-1 lists the arithmetic assignment operators. In each case, the righthand operand is an expression whose value is converted to a number.

Table 5-1. Arithmetic Assignment Operators

Operator	Effect
lvalue += *increment*	Adds *increment* to the value of *lvalue*.
lvalue -= *decrement*	Subtracts *decrement* from the value of *lvalue*.
lvalue *= *coefficient*	Multiplies the value of *lvalue* by *coefficient*.
lvalue /= *divisor*	Divides the value of *lvalue* by *divisor*.
lvalue %= *modulus*	Sets *lvalue* to its remainder by *modulus*.
lvalue ^= *power*	Raises *lvalue* to the power *power*.
lvalue **= *power*	Raises *lvalue* to the power *power*.

Only the ^= operator is specified by POSIX. For maximum portability, do not use the **= operator.

Syntactic Ambiguities Between /= and Regular Expressions

There is a syntactic ambiguity between the /= assignment operator and regexp constants whose first character is an =. (d.c.) This is most notable in commercial *awk* versions. For example:

```
$ awk /==/ /dev/null
awk: syntax error at source line 1
 context is
         >>> /= <<<
awk: bailing out at source line 1
```

A workaround is:

```
awk '/[=]=/' /dev/null
```

gawk does not have this problem, nor do the other freely available versions described in the section "Other Freely Available awk Implementations" in Appendix B, *Installing gawk*.

Increment and Decrement Operators

Increment and *decrement operators* increase or decrease the value of a variable by one. An assignment operator can do the same thing, so the increment operators add no power to the *awk* language; however, they are convenient abbreviations for very common operations.

The operator used for adding one is written ++. It can be used to increment a variable either before or after taking its value. To pre-increment a variable v, write ++v. This adds one to the value of v—that new value is also the value of the expression. (The assignment expression `v += 1` is completely equivalent.) Writing the ++ after the variable specifies post-increment. This increments the variable value just the same; the difference is that the value of the increment expression itself is the variable's *old* value. Thus, if `foo` has the value four, then the expression `foo++` has the value four, but it changes the value of `foo` to five. In other words, the operator returns the old value of the variable, but with the side effect of incrementing it.

The post-increment `foo++` is nearly the same as writing `(foo += 1) - 1`. It is not perfectly equivalent because all numbers in *awk* are floating-point—in floating-point, `foo + 1 - 1` does not necessarily equal `foo`. But the difference is minute as long as you stick to numbers that are fairly small (less than 10^{12}).

Fields and array elements are incremented just like variables. (Use `$(i++)` when you want to do a field reference and a variable increment at the same time. The parentheses are necessary because of the precedence of the field reference operator `$`.)

The decrement operator -- works just like ++, except that it subtracts one instead of adding it. As with ++, it can be used before the lvalue to pre-decrement or after it to post-decrement. Following is a summary of increment and decrement expressions:

`++`*`lvalue`*
: This expression increments *lvalue*, and the new value becomes the value of the expression.

`lvalue``++`
: This expression increments *lvalue*, but the value of the expression is the *old* value of *lvalue*.

`--`*`lvalue`*
: This expression is like ++*lvalue*, but instead of adding, it subtracts. It decrements *lvalue* and delivers the value that is the result.

lvalue--
: This expression is like *lvalue*++, but instead of adding, it subtracts. It decrements *lvalue*. The value of the expression is the *old* value of *lvalue*.

Operator Evaluation Order

What happens for something like the following?

```
b = 6
print b += b++
```

Or something even stranger?

```
b = 6
b += ++b + b++
print b
```

In other words, when do the various side effects prescribed by the postfix operators (`b++`) take effect? When side effects happen is *implementation defined*. In other words, it is up to the particular version of *awk*. The result for the first example may be 12 or 13, and for the second, it may be 22 or 23.

In short, doing things like this is not recommended and definitely not anything that you can rely upon for portability. You should avoid such things in your own programs.

True and False in awk

Many programming languages have a special representation for the concepts of "true" and "false." Such languages usually use the special constants `true` and `false`, or perhaps their uppercase equivalents. However, *awk* is different. It borrows a very simple concept of true and false from C. In *awk*, any nonzero numeric value *or* any nonempty string value is true. Any other value (zero or the null string `""`) is false. The following program prints `"A strange truth value"` three times:

```
BEGIN {
    if (3.1415927)
        print "A strange truth value"
    if ("Four Score And Seven Years Ago")
        print "A strange truth value"
    if (j = 57)
        print "A strange truth value"
}
```

There is a surprising consequence of the "nonzero or non-null" rule: the string constant `"0"` is actually true, because it is non-null. (d.c.)

Variable Typing and Comparison Expressions

Unlike other programming languages, *awk* variables do not have a fixed type. Instead, they can be either a number or a string, depending upon the value that is assigned to them.

The 1992 POSIX standard introduced the concept of a *numeric string*, which is simply a string that looks like a number—for example, `" +2"`. This concept is used for determining the type of a variable. The type of the variable is important because the types of two variables determine how they are compared. In *gawk*, variable typing follows these rules:

- A numeric constant or the result of a numeric operation has the *numeric* attribute.
- A string constant or the result of a string operation has the *string* attribute.
- Fields, `getline` input, `FILENAME`, `ARGV` elements, `ENVIRON` elements, and the elements of an array created by `split` that are numeric strings have the *strnum* attribute. Otherwise, they have the *string* attribute. Uninitialized variables also have the *strnum* attribute.
- Attributes propagate across assignments but are not changed by any use.

The last rule is particularly important. In the following program, `a` has numeric type, even though it is later used in a string operation:

```
BEGIN {
    a = 12.345
    b = a " is a cute number"
    print b
}
```

When two operands are compared, either string comparison or numeric comparison may be used. This depends upon the attributes of the operands, according to the following symmetric matrix:

	STRING	NUMERIC	STRNUM
STRING	string	string	string
NUMERIC	string	numeric	numeric
STRNUM	string	numeric	numeric

The basic idea is that user input that looks numeric—and *only* user input—should be treated as numeric, even though it is actually made of characters and is therefore also a string. Thus, for example, the string constant `" +3.14"` is a string, even though it looks numeric, and is *never* treated as number for comparison purposes.

In short, when one operand is a "pure" string, such as a string constant, then a string comparison is performed. Otherwise, a numeric comparison is performed.*

Comparison expressions compare strings or numbers for relationships such as equality. They are written using *relational operators*, which are a superset of those in C. Table 5-2 describes them.

Table 5-2. Relational Operators

Expression	Result
x < *y*	True if *x* is less than *y*.
x <= *y*	True if *x* is less than or equal to *y*.
x > *y*	True if *x* is greater than *y*.
x >= *y*	True if *x* is greater than or equal to *y*.
x == *y*	True if *x* is equal to *y*.
x != *y*	True if *x* is not equal to *y*.
x ~ *y*	True if the string *x* matches the regexp denoted by *y*.
x !~ *y*	True if the string *x* does not match the regexp denoted by *y*.
subscript in *array*	True if the array *array* has an element with the subscript *subscript*.

Comparison expressions have the value one if true and zero if false. When comparing operands of mixed types, numeric operands are converted to strings using the value of `CONVFMT` (see the section "Conversion of Strings and Numbers" earlier in this chapter).

Strings are compared by comparing the first character of each, then the second character of each, and so on. Thus, `"10"` is less than `"9"`. If there are two strings where one is a prefix of the other, the shorter string is less than the longer one. Thus, `"abc"` is less than `"abcd"`.

It is very easy to accidentally mistype the `==` operator and leave off one of the `=` characters. The result is still valid *awk* code, but the program does not do what is intended:

```
if (a = b)   # oops! should be a == b
    ...
else
    ...
```

Unless `b` happens to be zero or the null string, the `if` part of the test always succeeds. Because the operators are so similar, this kind of error is very difficult to spot when scanning the source code.

---

* The POSIX standard is under revision. The revised standard's rules for typing and comparison are the same as just described for *gawk*.

The following list illustrates the kind of comparison *gawk* performs, as well as what the result of the comparison is:

`1.5 <= 2.0`
: Numeric comparison (true)

`"abc" >= "xyz"`
: String comparison (false)

`1.5 != " +2"`
: String comparison (true)

`"1e2" < "3"`
: String comparison (true)

```
a = 2; b = "2"
a == b
```
: String comparison (true)

```
a = 2; b = " +2"
a == b
```
: String comparison (false)

In the next example:

```
$ echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
false
```

the result is `false` because both `$1` and `$2` are user input. They are numeric strings—therefore both have the *strnum* attribute, dictating a numeric comparison. The purpose of the comparison rules and the use of numeric strings is to attempt to produce the behavior that is "least surprising," while still "doing the right thing." String comparisons and regular expression comparisons are very different. For example:

```
x == "foo"
```

has the value one, or is true if the variable `x` is precisely `foo`. By contrast:

```
x ~ /foo/
```

has the value one if `x` contains `foo`, such as `"Oh, what a fool am I!"`.

The righthand operand of the `~` and `!~` operators may be either a regexp constant (`/.../`) or an ordinary expression. In the latter case, the value of the expression as a string is used as a dynamic regexp (see the section "How to Use Regular Expressions" and the section "Using Dynamic Regexps" in Chapter 2).

In modern implementations of *awk*, a constant regular expression in slashes by itself is also an expression. The regexp `/regexp/` is an abbreviation for the following comparison expression:

```
$0 ~ /regexp/
```

One special place where `/foo/` is *not* an abbreviation for `$0 ~ /foo/` is when it is the righthand operand of `~` or `!~`. See the section "Using Regular Expression Constants" earlier in this chapter, where this is discussed in more detail.

Boolean Expressions

A *Boolean expression* is a combination of comparison expressions or matching expressions, using the Boolean operators "or" (`||`), "and" (`&&`), and "not" (`!`), along with parentheses to control nesting. The truth value of the Boolean expression is computed by combining the truth values of the component expressions. Boolean expressions are also referred to as *logical expressions*. The terms are equivalent.

Boolean expressions can be used wherever comparison and matching expressions can be used. They can be used in `if`, `while`, `do`, and `for` statements (see the section "Control Statements in Actions" in Chapter 6). They have numeric values (one if true, zero if false) that come into play if the result of the Boolean expression is stored in a variable or used in arithmetic.

In addition, every Boolean expression is also a valid pattern, so you can use one as a pattern to control the execution of rules. The Boolean operators are:

boolean1 `&&` *boolean2*
: True if both *boolean1* and *boolean2* are true. For example, the following statement prints the current input record if it contains both `2400` and `foo`:

```
if ($0 ~ /2400/ && $0 ~ /foo/) print
```

The subexpression *boolean2* is evaluated only if *boolean1* is true. This can make a difference when *boolean2* contains expressions that have side effects. In the case of `$0 ~ /foo/ && ($2 == bar++)`, the variable `bar` is not incremented if there is no substring `foo` in the record.

boolean1 `||` *boolean2*
: True if at least one of *boolean1* or *boolean2* is true. For example, the following statement prints all records in the input that contain *either* `2400` or `foo` or both:

```
if ($0 ~ /2400/ || $0 ~ /foo/) print
```

The subexpression *boolean2* is evaluated only if *boolean1* is false. This can make a difference when *boolean2* contains expressions that have side effects.

`!` *boolean*

True if *boolean* is false. For example, the following program prints `no home!` in the unusual event that the HOME environment variable is not defined:

```
BEGIN { if (! ("HOME" in ENVIRON))
              print "no home!" }
```

(The `in` operator is described in the section "Referring to an Array Element" in Chapter 7.)

The `&&` and `||` operators are called *short-circuit* operators because of the way they work. Evaluation of the full expression is "short-circuited" if the result can be determined part way through its evaluation.

Statements that use `&&` or `||` can be continued simply by putting a newline after them. But you cannot put a newline in front of either of these operators without using backslash continuation (see the section "awk Statements Versus Lines" in Chapter 1, *Getting Started with awk*).

The actual value of an expression using the `!` operator is either one or zero, depending upon the truth value of the expression it is applied to. The `!` operator is often useful for changing the sense of a flag variable from false to true and back again. For example, the following program is one way to print lines in between special bracketing lines:

```
$1 == "START"   { interested = ! interested; next }
interested == 1 { print }
$1 == "END"     { interested = ! interested; next }
```

The variable `interested`, as with all *awk* variables, starts out initialized to zero, which is also false. When a line is seen whose first field is `START`, the value of `interested` is toggled to true, using `!`. The next rule prints lines as long as `interested` is true. When a line is seen whose first field is `END`, `interested` is toggled back to false.

The `next` statement is discussed in the section "The next Statement" in Chapter 6. `next` tells *awk* to skip the rest of the rules, get the next record, and start processing the rules over again at the top. The reason it's there is to avoid printing the bracketing `START` and `END` lines.

Conditional Expressions

A *conditional expression* is a special kind of expression that has three operands. It allows you to use one expression's value to select one of two other expressions. The conditional expression is the same as in the C language, as shown here:

```
selector ? if-true-exp : if-false-exp
```

There are three subexpressions. The first, *selector*, is always computed first. If it is "true" (not zero or not null), then *if-true-exp* is computed next and its value becomes the value of the whole expression. Otherwise, *if-false-exp* is computed next and its value becomes the value of the whole expression. For example, the following expression produces the absolute value of `x`:

```
x >= 0 ? x : -x
```

Each time the conditional expression is computed, only one of *if-true-exp* and *if-false-exp* is used; the other is ignored. This is important when the expressions have side effects. For example, this conditional expression examines element `i` of either array `a` or array `b`, and increments `i`:

```
x == y ? a[i++] : b[i++]
```

This is guaranteed to increment `i` exactly once, because each time only one of the two increment expressions is executed and the other is not. See Chapter 7 for more information about arrays.

As a minor *gawk* extension, a statement that uses `?:` can be continued simply by putting a newline after either character. However, putting a newline in front of either character does not work without using backslash continuation (see the section "awk Statements Versus Lines" in Chapter 1). If *––posix* is specified (see the section "Command-Line Options" in Chapter 11), then this extension is disabled.

Function Calls

A *function* is a name for a particular calculation. This enables you to ask for it by name at any point in the program. For example, the function `sqrt` computes the square root of a number.

A fixed set of functions are *built-in*, which means they are available in every *awk* program. The `sqrt` function is one of these. See the section "Built-in Functions" in Chapter 8 for a list of built-in functions and their descriptions. In addition, you can define functions for use in your program. See the section "User-Defined Functions" in Chapter 8 for instructions on how to do this.

The way to use a function is with a *function call* expression, which consists of the function name followed immediately by a list of *arguments* in parentheses. The

arguments are expressions that provide the raw materials for the function's calculations. When there is more than one argument, they are separated by commas. If there are no arguments, just write `()` after the function name. The following examples show function calls with and without arguments:

```
sqrt(x^2 + y^2)        # one argument
atan2(y, x)            # two arguments
rand()                 # no arguments
```

Do not put any space between the function name and the open parenthesis! A user-defined function name looks just like the name of a variable—a space would make the expression look like concatenation of a variable with an expression inside parentheses.

With built-in functions, space before the parenthesis is harmless, but it is best not to get into the habit of using space to avoid mistakes with user-defined functions. Each function expects a particular number of arguments. For example, the `sqrt` function must be called with a single argument, the number of which to take the square root:

```
sqrt(argument)
```

Some of the built-in functions have one or more optional arguments. If those arguments are not supplied, the functions use a reasonable default value. See the section "Built-in Functions" in Chapter 8 for full details. If arguments are omitted in calls to user-defined functions, then those arguments are treated as local variables and initialized to the empty string (see the section "User-Defined Functions" in Chapter 8).

Like every other expression, the function call has a value, which is computed by the function based on the arguments you give it. In this example, the value of `sqrt(`*`argument`*`)` is the square root of *argument*. A function can also have side effects, such as assigning values to certain variables or doing I/O. The following program reads numbers, one number per line, and prints the square root of each one:

```
$ awk '{ print "The square root of", $1, "is", sqrt($1) }'
1
The square root of 1 is 1
3
The square root of 3 is 1.73205
5
The square root of 5 is 2.23607
Ctrl-d
```

Operator Precedence (How Operators Nest)

Operator precedence determines how operators are grouped when different operators appear close by in one expression. For example, `*` has higher precedence than `+`; thus, `a + b * c` means to multiply `b` and `c`, and then add `a` to the product (i.e., `a + (b * c)`).

The normal precedence of the operators can be overruled by using parentheses. Think of the precedence rules as saying where the parentheses are assumed to be. In fact, it is wise to always use parentheses whenever there is an unusual combination of operators, because other people who read the program may not remember what the precedence is in this case. Even experienced programmers occasionally forget the exact rules, which leads to mistakes. Explicit parentheses help prevent any such mistakes.

When operators of equal precedence are used together, the leftmost operator groups first, except for the assignment, conditional, and exponentiation operators, which group in the opposite order. Thus, `a - b + c` groups as `(a - b) + c` and `a = b = c` groups as `a = (b = c)`.

The precedence of prefix unary operators does not matter as long as only unary operators are involved, because there is only one way to interpret them: innermost first. Thus, `$++i` means `$(++i)` and `++$x` means `++($x)`. However, when another operator follows the operand, then the precedence of the unary operators can matter. `$x^2` means `($x)^2`, but `-x^2` means `-(x^2)`, because `-` has lower precedence than `^`, whereas `$` has higher precedence. This list presents *awk*'s operators, in order of highest to lowest precedence:

`(...)`
: Grouping.

`$`
: Field.

`++ --`
: Increment, decrement.

`^ **`
: Exponentiation. These operators group right to left.

`+ - !`
: Unary plus, minus, logical "not."

`* / %`
: Multiplication, division, modulus.

`+ -`

Addition, subtraction.

String Concatenation

No special symbol is used to indicate concatenation. The operands are simply written side by side (see the earlier section "String Concatenation").

`< <= == !=`
`> >= >> | |&`

Relational and redirection. The relational operators and the redirections have the same precedence level. Characters such as `>` serve both as relationals and as redirections; the context distinguishes between the two meanings.

Note that the I/O redirection operators in `print` and `printf` statements belong to the statement level, not to expressions. The redirection does not produce an expression that could be the operand of another operator. As a result, it does not make sense to use a redirection operator near another operator of lower precedence without parentheses. Such combinations (for example, `print foo > a ? b : c`), result in syntax errors. The correct way to write this statement is `print foo > (a ? b : c)`.

`~ !~`

Matching, nonmatching.

`in` Array membership.

`&&` Logical "and."

`||` Logical "or."

`?:` Conditional. This operator groups right to left.

`= += -= *=`
`/= %= ^= **=`

Assignment. These operators group right to left.

The `|&`, `**`, and `**=` operators are not specified by POSIX. For maximum portability, do not use them.