Go to the first, previous, next, last section, table of contents.

Control Statements in Actions

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

All the control statements start with special keywords such as `if` and `while`, to distinguish them from simple expressions.

Many control statements contain other statements; for example, the `if` statement contains another statement which may or may not be executed. The contained statement is called the **body**. If you want to include more than one statement in the body, group them into a single **compound statement** with curly braces, separating them with newlines or semicolons.

- If Statement: Conditionally execute some `awk` statements.
- While Statement: Loop until some condition is satisfied.
- Do Statement: Do specified action while looping until some condition is satisfied.
- For Statement: Another looping statement, that provides initialization and increment clauses.
- Break Statement: Immediately exit the innermost enclosing loop.
- Continue Statement: Skip to the end of the innermost enclosing loop.
- Next Statement: Stop processing the current input record.
- Nextfile Statement: Stop processing the current file.
- Exit Statement: Stop execution of `awk`.

The `if-else` Statement

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

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

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

Here is an example:

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

In this example, if the expression `` `x % 2 == 0' `` is true (that is, the value of `x` is evenly divisible by two), then the first `print` statement is executed, otherwise the second `print` statement is executed.

If the `else` appears on the same line as *then-body*, and *then-body* is not a compound statement (i.e. not surrounded by curly braces), then a semicolon must separate *then-body* from `else`. To illustrate this, let's rewrite the previous example:

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

If you forget the `;', awk won't be able to interpret the statement, and you will get a syntax error.

We would not actually write this example this way, because a human reader might fail to see the else if it were not the first thing on its line.

The while Statement

In programming, a **loop** means a part of a program that can be executed two or more times in succession.

The while statement is the simplest looping statement in awk. It repeatedly executes a statement as long as a condition is true. It looks like this:

```
while (condition)
  body
```

Here *body* is a statement that we call the **body** of the loop, and *condition* is an expression that controls how long the loop keeps running.

The first thing the while statement does is test *condition*. If *condition* is true, it executes the statement *body*. After *body* has been executed, *condition* is tested again, and if it is still true, *body* is executed again. This process repeats until *condition* is no longer true. If *condition* is initially false, the body of the loop is never executed, and awk continues with the statement following the loop.

This example prints the first three fields of each record, one per line.

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

Here the body of the loop is a compound statement enclosed in braces, containing two statements.

The loop works like this: first, the value of i is set to one. Then, the while tests whether i is less than or equal to three. This is true when i equals one, so the i-th field is printed. Then the `i++' increments the value of i and the loop repeats. The loop terminates when i reaches four.

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

The do-while Statement

The do loop is a variation of the while looping statement. The do loop executes the *body* once, and then

repeats *body* as long as *condition* is true. It looks like this:

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

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

```
while (condition)
  body
```

This statement does not execute *body* even once if *condition* is false to begin with.

Here is an example of a `do` statement:

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

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

The `for` Statement

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

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

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

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

Here is an example of a `for` statement:

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

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

You cannot set more than one variable in the *initialization* part unless you use a multiple assignment statement such as `` `x = y = 0' ``, which is possible only if all the initial values are equal. (But you can

initialize additional variables by writing their assignments as separate statements preceding the `for` loop.)

The same is true of the *increment* part; to increment additional variables, you must write separate statements at the end of the loop. The C compound expression, using C's comma operator, would be useful in this context, but it is not supported in `awk`.

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

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

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

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

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

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

There is an alternate version of the `for` loop, for iterating over all the indices of an array:

```
for (i in array)
    do something with array[i]
```

See section Scanning All Elements of an Array, for more information on this version of the `for` loop.

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

The next section has more complicated examples of `for` loops.

The `break` Statement

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

```
awk '# find smallest divisor of num
     { num = $1
```

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

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

Here is another program equivalent to the previous one. It illustrates how the *condition* of a `for` or `while` could just as well be replaced with a `break` inside an `if`:

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

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

The `continue` Statement

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

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

```
awk 'BEGIN {
     for (x = 0; x <= 20; x++) {
         if (x == 5)
             continue
```

```
        printf "%d ", x
    }
    print ""
}'
```

This program prints all the numbers from zero to 20, except for five, for which the `printf` is skipped. Since the increment `` `x++' `` is not skipped, `x` does not remain stuck at five. Contrast the `for` loop above with this `while` loop:

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

This program loops forever once `x` gets to five.

As described above, the `continue` statement has no meaning when used outside the body of a loop. However, although it was never documented, historical implementations of `awk` have treated the `continue` statement outside of a loop as if it were a `next` statement (see section The `next` Statement). Recent versions of Unix `awk` no longer allow this usage. `gawk` will support this use of `continue` only if `` `--traditional' `` has been specified on the command line (see section Command Line Options). Otherwise, it will be treated as an error, since the POSIX standard specifies that `continue` should only be used inside the body of a loop (d.c.).

The `next` Statement

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

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

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

For example, if your `awk` program works only on records with four fields, and you don't want it to fail when given bad input, you might use this rule near the beginning of the program:

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

```
}
```

so that the following rules will not see the bad record. The error message is redirected to the standard error output stream, as error messages should be. See section Special File Names in gawk.

According to the POSIX standard, the behavior is undefined if the next statement is used in a BEGIN or END rule. gawk will treat it as a syntax error. Although POSIX permits it, some other awk implementations don't allow the next statement inside function bodies (see section User-defined Functions). Just as any other next statement, a next inside a function body reads the next record and starts processing it with the first rule in the program.

If the next statement causes the end of the input to be reached, then the code in any END rules will be executed. See section The BEGIN and END Special Patterns.

The nextfile Statement

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

Upon execution of the nextfile statement, FILENAME is updated to the name of the next data file listed on the command line, FNR is reset to one, ARGIND is incremented, and processing starts over with the first rule in the progam. See section Built-in Variables.

If the nextfile statement causes the end of the input to be reached, then the code in any END rules will be executed. See section The BEGIN and END Special Patterns.

The nextfile statement is a gawk extension; it is not (currently) available in any other awk implementation. See section Implementing nextfile as a Function, for a user-defined function you can use to simulate the nextfile statement.

The nextfile statement would be useful if you have many data files to process, and you expect that you would not want to process every record in every file. Normally, in order to move on to the next data file, you would have to continue scanning the unwanted records. The nextfile statement accomplishes this much more efficiently.

Caution: Versions of gawk prior to 3.0 used two words (`next file') for the nextfile statement. This was changed in 3.0 to one word, since the treatment of `file' was inconsistent. When it appeared after next, it was a keyword. Otherwise, it was a regular identifier. The old usage is still accepted. However, gawk will generate a warning message, and support for next file will eventually be discontinued in a future version of gawk.

The exit Statement

The exit statement causes awk to immediately stop executing the current rule and to stop processing input; any remaining input is ignored. It looks like this:

```
exit [return code]
```

If an `exit` statement is executed from a `BEGIN` rule the program stops processing everything immediately. No input records are read. However, if an `END` rule is present, it is executed (see section The `BEGIN` and `END` Special Patterns).

If `exit` is used as part of an `END` rule, it causes the program to stop immediately.

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

If you do not want the `END` rule to do its job in this case, you can set a variable to non-zero before the `exit` statement, and check that variable in the `END` rule. See section Assertions, for an example that does this.

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

For example, let's say you've discovered an error condition you really don't know how to handle. Conventionally, programs report this by exiting with a non-zero status. Your `awk` program can do this using an `exit` statement with a non-zero argument. Here is an example:

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

Go to the first, previous, next, last section, table of contents.