Mastering Modern Linux by Paul S. Wang

Appendix: Pattern Processing with `awk`

The **awk** program is a powerful yet simple filter. It processes its input one line at a time, applying user-specified **awk** *pattern actions* to each line. The **awk** program is similar to, but more powerful than, **sed**. The **awk** mechanisms are based more on the C programming language than on a text editor, allowing for variables, arrays, conditionals, expressions, iteration controls, formatted output, and so on. The **awk** program can perform operations not possible with **sed**, such as joining adjacent lines and comparing parts of different lines.

The general form of the **awk** command is

awk [`-F`*c*] *script* [*file*] ...

The `-F` option specifies a character *c* to be the *field separator* (default white space). The argument *script* is an **awk** script given on the command line or in a file with the `-f` *filename* convention. The files are are processed in the order given. If no files are given, standard input is used. If a dash (`-`) is given as a file name, it is taken to mean standard input.

The **awk** processing cycle is as follows:

1. If there are no more input lines, terminate. Otherwise, read the next input line.
2. Apply all **awk** pattern commands sequentially as specified in *script* to the current line.
3. Go to step (1).

Note that unlike **sed**, **awk** does not write lines to the standard output automatically.

An **awk** script consists of one or more pattern actions given on different lines or separated by semicolons. Each pattern action takes the form

pattern `{`*action*`}`

If the current line matches the *pattern*, the *action* is taken. A missing pattern matches every line, and a missing action outputs the line. Thus,

```
ls -l | awk '/Linux/'
```

is the same as

```
ls -l | sed -n '/Linux/p'
```

Pattern and *action* are described more fully in the following subsections.

The concept of a field here is the same as that used for **sort**: **awk** delineates each of its input lines into fields separated by white space or by a field separator character specified with the `-F` option. In an **awk** action, the fields are denoted `$1`, `$2`, and so on. The entire line is denoted by `$0`.

While it is hard to rearrange the order of fields using **sed**, it is easy with **awk**. For instance, the output of **ls** `-l` has eight fields:

```
-rw-rw---- 2 jsmith 512 Apr 23 21:44 report.tex
-rw-rw---- 1 jsmith 79 Feb 9 15:13 Makefile
-rw-rw---- 2 jsmith 1024 Feb 25 00:13 pipe.c
```

When the preceding lines are piped through **awk**,

ls -l | awk '{print $8,$4,$5,$6}'

The following output is produced:

```
report.tex   512 Apr 23
Makefile      79 Feb 9
pipe.c      1024 Feb 25
```

Note that, in this example, the pattern action contains no pattern. Also, actions are always enclosed between braces ({ and }).

awk Patterns

As with **sed**, the *pattern* determines whether or not **awk** takes an action on the current line. In fact, a **sed** address, specified with one or two match expressions, also will work as an **awk** pattern. If you are familiar with **sed**, you already know many useful patterns. For instance,

/^first/	(first at the beginning of a line)
/last$/	(last at the end of a line)
/^$/	(an empty line)
/[␣▷][␣▷]*/	(a line with a string of one or more blanks)
/*begin*/,/*end*/	(all lines between a *begin* match and an *end* match)

are valid patterns in both **sed** and **awk**.

In **awk**, a pattern is an arbitrary Boolean expression involving *regular expressions* and *relational expressions*. Boolean expressions are formed with && (and), || (or), ! (not), and parentheses. A regular expression in **awk** must begin and end with a slash (/) and otherwise is defined the same as that for **egrep** (Table **??**). Relational expressions are formed using C-like operators >, >= , <, <=, == (equal), and != (not equal). In addition, a relational expression can be:

expression ~ *re*
expression !~ *re*

where ~ means "contains" and !~ means "does not contain." For example, the pattern

```
$1 ~ /GNU/ && $2 ~ /Linux/
```

is true if the first field contains the string GNU and the second field contains the string Linux.

A pattern may contain two patterns separated by a comma, in which case the action is applied to all lines beginning with a line matching the first pattern up to and including the line matching the second pattern (the same as in **sed**). Thus,

awk 'NR==14,NR==30' *file*

outputs lines 14-30 of *file*, because **awk** keeps a running line count in the built-in variable NR. Other useful built-in variables are listed in Table 0.2.

The special patterns BEGIN and END in

BEGIN {*action*}
END {*action*}

specify actions executed before the first input line and after the last input line, respectively. They are used for initialization and postprocessing when needed.

TABLE 0.2 Built-in `awk` Variables

Variable	Meaning
`NF`	Total number of fields on current line
`NR`	Sequence number of current line
`FS`	Input field separator character (default blanks)
`RS`	Input record separator (default NEWLINE)
`OFS`	Output field separator string (default SPACE)
`ORS`	Output record separator string (default NEWLINE)
`OFMT`	Output format for numbers (default `%g` as in `printf`)

`awk` Actions

Now let's turn to the question of how *actions* are specified. An *action* contains a sequence of statements given on different lines or separated by semicolons. Possible statements are as follows:

Assignment:	*var* `=` *expression*
Output:	`print` *expression* [, *expression*] ...
	`printf(` ... `)` (`as in C`)
Flow control (as in C):	`if` (*conditional*) *statement* [`else` *statement*]
	`for`(*expression*`;` *conditional*`;` *expression*) *statement*
	`while` (*conditional*) *statement*, `break`, `continue`
Additional flow control:	`next` (skip remaining commands, start next **awk** cycle)
	`exit` (exit **awk**)

In the preceding definitions, a *statement* can be a compound statement in the form

`{`*statement*, *statement*, ... `}`

The output statements use the standard output. However, they can be followed by `>` `"`*filename*`"` to redirect the output into a file.

`awk` Expressions

Expressions in **awk** statements can be constants, variables, arrays, fields, or any combinations of these using the following C operators:

`+, -, *, /, %, ++, --, +=, -=, *=, %=`

Numerical constants in **awk** statements are the same as in C. String constants are placed in double quotation marks (`"`*string*`"`) and variables are initialized to the null string. An array element is denoted as `a[`*i*`]`, where *i* can be an integer or any string. A blank between two expressions concatenates them into a string. Thus, for example,

awk `'{print $2 ":" $1}'` *file*

outputs *field2*:*field1* of each line from the given file. Built-in functions (Table 0.3 lists a few) can also be used in expressions. In **awk**, conditional expressions use C notation and may involve **awk**-defined relational expressions. In Table 0.3, e is an expression, c is a character, s is a string, and i and j are integers.

Index Preparation: An Example

The **awk** pattern processing program is powerful and involved. The best way to learn it is through use and experimentation. In this section, we present an example of **awk** usage to prepare an index for a document (**Ex:** `ex04/index.awk`). Suppose you have several index files, each containing entries such as (**Ex:** `ex04/index.data`)

```
bash:99
regular expression:155
bash:123
pipe:101
gnome:163
socket:415
pipe:23
```

where each line has two fields: an index item and a page number separated by the `:`. Your goal is to produce an overall index file in alphabetical order with lines such as (**Ex:** `ex04/index.file`)

```
bash 99,123
gnome 163
pipe 23,101
regular expression 155
socket 415
```

The first step is to order the entries alphabetically and by page number, which can be done with

```
sort -t: --key=1,2.0f --key=2n index.data >| index.tmp
```

in which the following sort keys are used:

`1,2.0f`	(first key, field one ignoring case)
`2n`	(second key, field two with numerical comparison)

It then remains to collect repeated index items to form lines with multiple page numbers. Since repeated items will be on consecutive lines, the **awk** script `index.awk` (Figure 4) can be used. To apply the script use

```
awk -f index.awk index.tmp >| index.file
```

There are four pattern commands in `index.awk`. The first command sets the variable `i` (used for initialization) to zero. The second command compares `$1` with the variable pre,

TABLE 0.3 Built-in **awk** Functions

Function	Meaning
`int(`*e*`)`, `length(`*s*`)`	Integer (floor), length of string *s*
`gsub(`*re*`, `*s*`, `*t*`)`	Replaces matches of *re* in *t* with *s*
`index(`*s1*`,`*s2*`)`	Position of string *s2* in *s1*, zero if *s2* not in *s1*
`sprintf(...)`	Format conversion, same as in the C language
`substr(`*s*`,`*i*`,`*j*`)`	Substring of *s* of length *j* from position *i*
`split(`*s*`,`*a*`,`*c*`)`	Cuts *s* into substrings *a*`[1]` to *a*`[`*i*`]` at char *c*; returns *i*
`getline()`	Inputs next line, returns 0 on end of file, otherwise 1

which stands for the previous index item and is initially null. If `$1` is equal to `pre` (field one is the same as the previous index item), then output the page number (`$2`), preceded by a comma. If `$1` is not equal to `pre` (a new index item), then output NEWLINE, `$1`, SPACE, and `$2` except for the very first line where the leading NEWLINE is not needed. The conditional output is performed in the **if** of the third command which also records the index item (`$1`) in the variable pre. At the end of the input file, a final NEWLINE is output.

FIGURE 4 Program `index.awk` for Index Processing

```
BEGIN {  i = 0; }

$1 == pre {  printf(",%s", $2); }

$1 != pre {  if (i > 0)
             {   printf("\n%s %s",$1,$2); }
             else
             {   printf("%s %s",$1,$2); i = 1; }
             pre = $1;
          }

END {  printf("\n"); }
```