Command: **awk – pattern matching language**
Syntax: **awk** *rules* [*file*] ...
Flags: (none)
Examples:
awk rules input # Process *input* according to *rules*
awk rules – >out # Input from terminal, output to *out*

AWK is a programming language devised by Aho, Weinberger, and Kernighan at Bell Labs (hence the name). *Awk* programs search files for specific patterns and performs ‘‘actions’’ for every occurrence of these patterns. The patterns can be ‘‘regular expressions’’ as used in the *ed* editor. The actions are expressed using a subset of the C language.

The patterns and actions are usually placed in a ‘‘rules’’ file whose name must be the first argument in the command line, preceded by the flag **–f**. Otherwise, the first argument on the command line is taken to be a string containing the rules themselves. All other arguments are taken to be the names of text files on which the rules are to be applied, with **–** being the standard input. To take rules from the standard input, use **–f –**.

The command:

```
awk rules prog.*
```

would read the patterns and actions rules from the file *rules* and apply them to all the arguments.

The general format of a rules file is:

<pattern> { <action> } <pattern> { <action> } ...

There may be any number of these <pattern> { <action> } sequences in the rules file. *Awk* reads a line of input from the current input file and applies every <pattern> { <action> } in sequence to the line.

If the <pattern> corresponding to any { <action> } is missing, the action is applied to every line of input. The default { <action> } is to print the matched input line.

Patterns

The <pattern>s may consist of any valid C expression. If the <pattern> consists of two expressions separated by a comma, it is taken to be a range and the <action> is performed on all lines of input that match the range. <pattern>s may contain ‘‘regular expressions’’ delimited by an @ symbol. Regular expressions can be thought of as a generalized ‘‘wildcard’’ string matching mechanism, similar to that used by many operating systems to specify file names. Regular expressions may contain any of the following characters:

x	An ordinary character
\	The backslash quotes any character
^	A circumflex at the beginning of an expr matches the beginning of a line.
$	A dollar-sign at the end of an expression matches the end of a line.

. A period matches any single character except newline.

* An expression followed by an asterisk matches zero or more occurrences of that expression: ‘‘fo*’’ matches ‘‘f’’, ‘‘fo’’, ‘‘foo’’, ‘‘fooo’’, etc.

+ An expression followed by a plus sign matches one or more occurrences of that expression: ‘‘fo+’’ matches ‘‘fo’’, ‘‘foo’’, ‘‘fooo’’, etc.

[] A string enclosed in square brackets matches any single character in that string, but no others. If the first character in the string is a circumflex, the expression matches any character except newline and the characters in the string. For example, ‘‘[xyz]’’ matches ‘‘xx’’ and ‘‘zyx’’, while ‘‘[ˆxyz]’’ matches ‘‘abc’’ but not ‘‘axb’’. A range of characters may be specified by two characters separated by ‘‘-’’.

Actions

Actions are expressed as a subset of the C language. All variables are global and default to int’s if not formally declared. Only char’s and int’s and pointers and arrays of char and int are allowed. *Awk* allows only decimal integer constants to be used—no hex (0xnn) or octal (0nn). String and character constants may contain all of the special C escapes (\n, \r, etc.).

Awk supports the ‘‘if’’, ‘‘else’’, ‘‘while’’ and ‘‘break’’ flow of control constructs, which behave exactly as in C.

Also supported are the following unary and binary operators, listed in order from highest to lowest precedence:

<table>
<tr><th>Operator</th><th>Type</th><th>Associativity</th></tr>
<tr><td>() []</td><td>unary</td><td>left to right</td></tr>
<tr><td>! ˜ ++ —– * &</td><td>unary</td><td>right to left</td></tr>
<tr><td>* / %</td><td>binary</td><td>left to right</td></tr>
<tr><td>+ –</td><td>binary</td><td>left to right</td></tr>
<tr><td><< >></td><td>binary</td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>binary</td><td>left to right</td></tr>
<tr><td>== !=</td><td>binary</td><td>left to right</td></tr>
<tr><td>&</td><td>binary</td><td>left to right</td></tr>
<tr><td>ˆ</td><td>binary</td><td>left to right</td></tr>
<tr><td>|</td><td>binary</td><td>left to right</td></tr>
<tr><td>&&</td><td>binary</td><td>left to right</td></tr>
<tr><td>||</td><td>binary</td><td>left to right</td></tr>
<tr><td>=</td><td>binary</td><td>right to left</td></tr>
</table>

Comments are introduced by a ’#’ symbol and are terminated by the first newline character. The standard ‘‘/*’’ and ‘‘*/’’ comment delimiters are not supported and will result in a syntax error.

Fields

When *awk* reads a line from the current input file, the record is automatically separated into ‘‘fields.’’ A field is simply a string of consecutive characters delimited by either the beginning or end of line, or a ‘‘field separator’’ character. Initially, the field separators are the space and tab character. The special unary operator ’$’ is used to reference one of the fields in the current input record (line). The fields are numbered sequentially starting at 1. The expression ‘‘$0’’ references the entire input line.

Similarly, the ‘‘record separator’’ is used to determine the end of an input ‘‘line,’’ initially the newline character. The field and record separators may be changed programatically by one of the actions and will remain in effect until changed again.

Multiple (up to 10) field separators are allowed at a time, but only one record separator.

Fields behave exactly like strings; and can be used in the same context as a character array. These ‘‘arrays’’ can be considered to have been declared as:

```
char ($n)[ 128 ];
```

In other words, they are 128 bytes long. Notice that the parentheses are necessary because the operators [] and $ associate from right to left; without them, the statement would have parsed as:

```
char $(1[ 128 ]);
```

which is obviously ridiculous.

If the contents of one of these field arrays is altered, the ‘‘$0’’ field will reflect this change. For example, this expression:

```
*$4 = ’A’;
```

will change the first character of the fourth field to an upper- case letter ’A’. Then, when the following input line:

```
120 PRINT "Name      address      Zip"
```

is processed, it would be printed as:

```
120 PRINT "Name      Address      Zip"
```

Fields may also be modified with the strcpy() function (see below). For example, the expression:

```
strcpy( $4, "Addr." );
```

applied to the same line above would yield:

```
120 PRINT "Name        Addr.       Zip"
```

Predefined Variables

The following variables are pre-defined:

FS	Field separator (see below).
RS	Record separator (see below also).
NF	Number of fields in current input record (line).
NR	Number of records processed thus far.
FILENAME	Name of current input file.
BEGIN	A special <pattern> that matches the beginning of input text.
END	A special <pattern> that matches the end of input text.

Awk also provides some useful built-in functions for string manipulation and printing:

print(arg)	Simple printing of strings only, terminated by ’\n’.
printf(arg...)	Exactly the printf() function from C.
getline()	Reads the next record and returns 0 on end of file.
nextfile()	Closes the current input file and begins processing the next file
strlen(s)	Returns the length of its string argument.
strcpy(s,t)	Copies the string ‘‘t’’ to the string ‘‘s’’.
strcmp(s,t)	Compares the ‘‘s’’ to ‘‘t’’ and returns 0 if they match.
toupper(c)	Returns its character argument converted to upper-case.
tolower(c)	Returns its character argument converted to lower-case.
match(s,@re@)	Compares the string ‘‘s’’ to the regular expression ‘‘re’’ and returns the number of matches found (zero if none).

Authors

Awk was written by Saeko Hirabauashi and Kouichi Hirabayashi.