



**NAME**

awk – pattern-directed scanning and processing language

**SYNOPSIS**

**awk** [ **-F** *fs* ] [ **-v** *var=value* ] [ '*prog*' | **-f** *progfile* ] [ *file* ... ]

**DESCRIPTION**

*Awk* scans each input *file* for lines that match any of a set of patterns specified literally in *prog* or in one or more files specified as **-f** *progfile*. With each pattern there can be an associated action that will be performed when a line of a *file* matches the pattern. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern. The file name **-** means the standard input. Any *file* of the form *var=value* is treated as an assignment, not a filename, and is executed at the time it would have been opened if it were a filename. The option **-v** followed by *var=value* is an assignment to be done before *prog* is executed; any number of **-v** options may be present. The **-F** *fs* option defines the input field separator to be the regular expression *fs*.

An input line is normally made up of fields separated by white space, or by regular expression **FS**. The fields are denoted **\$1**, **\$2**, ..., while **\$0** refers to the entire line. If **FS** is null, the input line is split into one field per character.

A pattern-action statement has the form

```
pattern { action }
```

A missing { *action* } means print the line; a missing pattern always matches. Pattern-action statements are separated by newlines or semicolons.

An action is a sequence of statements. A statement can be one of the following:

```
if( expression ) statement [ else statement ]
while( expression ) statement
for( expression ; expression ; expression ) statement
for( var in array ) statement
do statement while( expression )
break
continue
{ [ statement ... ] }
expression                # commonly var = expression
print [ expression-list ] [ > expression ]
printf format [ , expression-list ] [ > expression ]
return [ expression ]
next                       # skip remaining patterns on this input line
nextfile                   # skip rest of this file, open next, start at top
delete array[ expression ] # delete an array element
delete array                # delete all elements of array
exit [ expression ]        # exit immediately; status is expression
```

Statements are terminated by semicolons, newlines or right braces. An empty *expression-list* stands for **\$0**. String constants are quoted " ", with the usual C escapes recognized within. Expressions take on string or numeric values as appropriate, and are built using the operators + - \* / % ^ (exponentiation), and concatenation (indicated by white space). The operators ! ++ -- += -= \*= /= %= ^= > >= < <= == != ?: are also available in expressions. Variables may be scalars, array elements (denoted *x[i]*) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. Multiple subscripts such as [**i,j,k**] are permitted; the constituents are concatenated, separated by the value of **SUBSEP**.

The **print** statement prints its arguments on the standard output (or on a file if *>file* or *>>file* is present or on a pipe if *|cmd* is present), separated by the current output field separator, and terminated by the output record separator. *file* and *cmd* may be literal names or parenthesized expressions; identical string values in

different statements denote the same open file. The **printf** statement formats its expression list according to the format (see *printf(3)*). The built-in function **close**(*expr*) closes the file or pipe *expr*. The built-in function **fflush**(*expr*) flushes any buffered output for the file or pipe *expr*.

The mathematical functions **exp**, **log**, **sqrt**, **sin**, **cos**, and **atan2** are built in. Other built-in functions:

**length** the length of its argument taken as a string, or of **\$0** if no argument.

**rand** random number on (0,1)

**srand** sets seed for **rand** and returns the previous seed.

**int** truncates to an integer value

**substr**(*s, m, n*)

the *n*-character substring of *s* that begins at position *m* counted from 1.

**index**(*s, t*)

the position in *s* where the string *t* occurs, or 0 if it does not.

**match**(*s, r*)

the position in *s* where the regular expression *r* occurs, or 0 if it does not. The variables **RSTART** and **RLENGTH** are set to the position and length of the matched string.

**split**(*s, a, fs*)

splits the string *s* into array elements *a*[1], *a*[2], ..., *a*[*n*], and returns *n*. The separation is done with the regular expression *fs* or with the field separator **FS** if *fs* is not given. An empty string as field separator splits the string into one array element per character.

**sub**(*r, t, s*)

substitutes *t* for the first occurrence of the regular expression *r* in the string *s*. If *s* is not given, **\$0** is used.

**gsub** same as **sub** except that all occurrences of the regular expression are replaced; **sub** and **gsub** return the number of replacements.

**sprintf**(*fmt, expr, ...*)

the string resulting from formatting *expr ...* according to the *printf(3)* format *fmt*

**system**(*cmd*)

executes *cmd* and returns its exit status

**tolower**(*str*)

returns a copy of *str* with all upper-case characters translated to their corresponding lower-case equivalents.

**toupper**(*str*)

returns a copy of *str* with all lower-case characters translated to their corresponding upper-case equivalents.

The “function” **getline** sets **\$0** to the next input record from the current input file; **getline <file** sets **\$0** to the next record from *file*. **getline x** sets variable *x* instead. Finally, *cmd* | **getline** pipes the output of *cmd* into **getline**; each call of **getline** returns the next line of output from *cmd*. In all cases, **getline** returns 1 for a successful input, 0 for end of file, and -1 for an error.

Patterns are arbitrary Boolean combinations (with **!**, **||**, and **&&**) of regular expressions and relational expressions. Regular expressions are as in *egrep*; see *grep(1)*. Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions, using the operators **~** and **!~**. */re/* is a constant regular expression; any string (constant or variable) may be used as a regular expression, except in the position of an isolated regular expression in a pattern.

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines from an occurrence of the first pattern though an occurrence of the second.

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
expression in array-name
(expr,expr,...) in array-name
```

where a relop is any of the six relational operators in C, and a matchop is either ~ (matches) or !~ (does not match). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns **BEGIN** and **END** may be used to capture control before the first input line is read and after the last. **BEGIN** and **END** do not combine with other patterns.

Variable names with special meanings:

#### **CONVFMT**

conversion format used when converting numbers (default **%.6g**)

**FS** regular expression used to separate fields; also settable by option **-Ffs**.

**NF** number of fields in the current record

**NR** ordinal number of the current record

**FNR** ordinal number of the current record in the current file

#### **FILENAME**

the name of the current input file

**RS** input record separator (default newline)

**OFS** output field separator (default blank)

**ORS** output record separator (default newline)

**OFMT** output format for numbers (default **%.6g**)

#### **SUBSEP**

separates multiple subscripts (default 034)

**ARGC** argument count, assignable

**ARGV** argument array, assignable; non-null members are taken as filenames

#### **ENVIRON**

array of environment variables; subscripts are names.

Functions may be defined (at the position of a pattern-action statement) thus:

```
function foo(a, b, c) { ...; return x }
```

Parameters are passed by value if scalar and by reference if array name; functions may be called recursively. Parameters are local to the function; all other variables are global. Thus local variables may be created by providing excess parameters in the function definition.

#### **EXAMPLES**

```
length($0) > 72
```

Print lines longer than 72 characters.

```
{ print $2, $1 }
```

Print first two fields in opposite order.

```
BEGIN { FS = ",[ \t]*|[ \t]+" }
{ print $2, $1 }
```

Same, with input fields separated by comma and/or blanks and tabs.

```
    { s += $1 }
END  { print "sum is", s, " average is", s/NR }
    Add up first column, print sum and average.

/start/, /stop/
    Print all lines between start/stop pairs.

BEGIN {      # Simulate echo(1)
    for (i = 1; i < ARGV; i++) printf "%s ", ARGV[i]
    printf "\n"
    exit }
```

**SEE ALSO**

*lex(1)*, *sed(1)*

A. V. Aho, B. W. Kernighan, P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, 1988.  
ISBN 0-201-07981-X

**BUGS**

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate " " to it. The scope rules for variables in functions are a botch; the syntax is worse.