

NAME

awk – pattern scanning and processing language

SYNOPSIS

`/usr/bin/awk [-f progfile] [-Fc] [' prog '] [parameters] [filename...]`

`/usr/xpg4/bin/awk [-FcERE] [-v assignment...] ' program ' -f progfile... [argument...]`

DESCRIPTION

The `/usr/xpg4/bin/awk` utility is described on the `nawk(1)` manual page.

The `/usr/bin/awk` utility scans each input *filename* for lines that match any of a set of patterns specified in *prog*. The *prog* string must be enclosed in single quotes (') to protect it from the shell. For each pattern in *prog* there can be an associated action performed when a line of a *filename* matches the pattern. The set of pattern-action statements can appear literally as *prog* or in a file specified with the `-f progfile` option. Input files are read in order; if there are no files, the standard input is read. The file name '-' means the standard input.

OPTIONS

The following options are supported:

`-f progfile` **awk** uses the set of patterns it reads from *progfile*.

`-Fc` Uses the character *c* as the field separator (FS) character. See the discussion of **FS** below.

USAGE**Input Lines**

Each input line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern. Any *filename* 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. *Variables* assigned in this manner are not available inside a **BEGIN** rule, and are assigned after previously specified files have been read.

An input line is normally made up of fields separated by white spaces. (This default can be changed by using the **FS** built-in variable or the `-Fc` option.) The default is to ignore leading blanks and to separate fields by blanks and/or tab characters. However, if **FS** is assigned a value that does not include any of the white spaces, then leading blanks are not ignored. The fields are denoted **\$1**, **\$2**, ...; **\$0** refers to the entire line.

Pattern-action Statements

A pattern-action statement has the form:

```
pattern { action }
```

Either pattern or action can be omitted. If there is no action, the matching line is printed. If there is no pattern, the action is performed on every input line. Pattern-action statements are separated by newlines or semicolons.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of relational expressions and regular expressions. A relational expression is one of the following:

```
expression relop expression
```

expression matchop regular_expression

where a *relop* is any of the six relational operators in C, and a *matchop* is either ~ (contains) or !~ (does not contain). An *expression* is an arithmetic expression, a relational expression, the special expression

var in array

or a Boolean combination of these.

Regular expressions are as in **egrep**(1). In patterns they must be surrounded by slashes. Isolated regular expressions in a pattern apply to the entire line. Regular expressions can also occur in relational expressions. A pattern can consist of two patterns separated by a comma; in this case, the action is performed for all lines between the occurrence of the first pattern to the occurrence of the second pattern.

The special patterns **BEGIN** and **END** can be used to capture control before the first input line has been read and after the last input line has been read respectively. These keywords do not combine with any other patterns.

Built-in Variables

Built-in variables include:

FILENAME	name of the current input file
FS	input field separator regular expression (default blank and tab)
NF	number of fields in the current record
NR	ordinal number of the current record
OFMT	output format for numbers (default %.6g)
OFS	output field separator (default blank)
ORS	output record separator (default new-line)
RS	input record separator (default new-line)

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

```

if ( expression ) statement [ else statement ]
while ( expression ) statement
do statement while ( expression )
for ( expression ; expression ; expression ) statement
for ( var in array ) statement
break
continue
{ [ statement ] ... }
expression      # commonly variable = expression
print [ expression-list ] [ >expression ]
printf format [ ,expression-list ] [ >expression ]
next             # skip remaining patterns on this input line
exit [expr]    # skip the rest of the input; exit status is expr

```

Statements are terminated by semicolons, newlines, or right braces. An empty expression-list stands for the whole input line. Expressions take on string or numeric values as appropriate, and are built using the operators +, -, *, /, %, ^ and concatenation (indicated by a blank). The operators ++, --, +=, -=, *=, /=, %=, ^=, >, >=, <, <=, ==, !=, and ?: are also available in expressions. Variables can be scalars, array elements (denoted *x*[*i*]), or fields. Variables are initialized to the null string or zero. Array subscripts can be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (""), with the usual C escapes recognized within.

The **print** statement prints its arguments on the standard output, or on a file if >*expression* is present, or on a pipe if '|*cmd*' is present. The output resulted from the print statement is terminated by the output record separator with each argument separated by the current output field separator. The **printf** statement formats its expression list according to the format (see **printf(3C)**).

Built-in Functions

The arithmetic functions are as follows:

cos (<i>x</i>)	Return cosine of <i>x</i> , where <i>x</i> is in radians. (In /usr/xpg4/bin/awk only. See nawk(1) .)
sin (<i>x</i>)	Return sine of <i>x</i> , where <i>x</i> is in radians. (In /usr/xpg4/bin/awk only. See nawk(1) .)
exp (<i>x</i>)	Return the exponential function of <i>x</i> .
log (<i>x</i>)	Return the natural logarithm of <i>x</i> .
sqrt (<i>x</i>)	Return the square root of <i>x</i> .

int(*x*) Truncate its argument to an integer. It is truncated toward **0** when $x > 0$.

The string functions are as follows:

index(*s*, *t*)

Return the position in string *s* where string *t* first occurs, or **0** if it does not occur at all.

int(*s*)

truncates *s* to an integer value. If *s* is not specified, \$0 is used.

length(*s*)

Return the length of its argument taken as a string, or of the whole line if there is no argument.

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

Split 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.

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

Format the expressions according to the **printf**(3C) format given by *fmt* and returns the resulting string.

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

returns the *n*-character substring of *s* that begins at position *m*.

The input/output function is as follows:

getline

Set **\$0** to the next input record from the current input file. **getline** returns **1** for successful input, **0** for end of file, and **-1** for an error.

Large File Behavior

See **largefile**(5) for the description of the behavior of **awk** when encountering files greater than or equal to 2 Gbyte (2^{31} bytes).

EXAMPLES**Example 1: Printing Lines Longer Than 72 Characters**

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It prints lines longer than seventy two characters:

```
length > 72
```

Example 2: Printing Fields in Opposite Order

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It prints the first two fields in opposite order:

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

Example 3: Printing Fields in Opposite Order with the Input Fields Separated

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It prints the first two input fields in opposite order, separated by a comma, blanks or tabs:

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

Example 4: Adding Up the First Column, Printing the Sum and Average

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It adds up the first column, and prints the sum and average:

```
{ s += $1 }  
END { print "sum is", s, " average is", s/NR }
```

Example 5: Printing Fields in Reverse Order

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It prints fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Example 6: Printing All lines Between start/stop Pairs

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It prints all lines between start/stop pairs.

```
/start/, /stop/
```

Example 7: Printing All Lines Whose First Field is Different from the Previous One

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It prints all lines whose first field is different from the previous one.

```
$1 != prev { print; prev = $1 }
```

Example 8: Printing a File and Filling in Page numbers

The following example is an **awk** script that can be executed by an **awk -f examplescript** style command. It prints a file and fills in page numbers starting at 5:

```
/Page/ { $2 = n++; }  
  { print }
```

Example 9: Printing a File and Numbering Its Pages

Assuming this program is in a file named **prog**, the following example prints the file **input** numbering its pages starting at 5:

```
example% awk -f prog n=5 input
```

ENVIRONMENT VARIABLES

See **environ(5)** for descriptions of the following environment variables that affect the execution of **awk**: **LANG**, **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **NLSPATH**, and **PATH**.

LC_NUMERIC Determine the radix character used when interpreting numeric input, performing conversions between numeric and string values and formatting numeric output. Regardless of locale, the period character (the decimal-point character of the POSIX locale) is the decimal-point character recognized in processing **awk** programs (including assignments in command-line arguments).

ATTRIBUTES

See **attributes(5)** for descriptions of the following attributes:

/usr/bin/awk

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE
TYPEATTRIBUTE VALUE AvailabilitySUNWesu CSINot Enabled

/usr/xpg4/bin/awk

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE
TYPEATTRIBUTE VALUE AvailabilitySUNWxcu4 CSIEnabled Interface StabilityStandard

SEE ALSO

egrep(1), **grep(1)**, **nawk(1)**, **sed(1)**, **printf(3C)**, **attributes(5)**, **environ(5)**, **largefile(5)**, **standards(5)**

NOTES

Input white space is not preserved on output if fields are involved.

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 an expression to be treated as a string, concatenate the null string ("") to it.