NAME

gawk − pattern scanning and processing language

SYNOPSIS

gawk [POSIX or GNU style options] **−f** *program-file* [**−−**] file …
gawk [POSIX or GNU style options] [**−−**] *program-text* file …

DESCRIPTION

Gawk is the GNU Project's implementation of the AWK programming language. It conforms to the definition of the language in the POSIX 1003.2 Command Language And Utilities Standard. This version in turn is based on the description in *The AWK Programming Language*, by Aho, Kernighan, and Weinberger, with the additional features found in the System V Release 4 version of UNIX *awk*. *Gawk* also provides more recent Bell Labs *awk* extensions, and some GNU-specific extensions.

The command line consists of options to *gawk* itself, the AWK program text (if not supplied via the **−f** or **−−file** options), and values to be made available in the **ARGC** and **ARGV** pre-defined AWK variables.

OPTION FORMAT

Gawk options may be either the traditional POSIX one letter options, or the GNU style long options. POSIX options start with a single "−", while long options start with "−−". Long options are provided for both GNU-specific features and for POSIX mandated features.

Following the POSIX standard, *gawk*-specific options are supplied via arguments to the **−W** option. Multiple **−W** options may be supplied, or multiple arguments may be supplied together if they are separated by commas, or enclosed in quotes and separated by white space. Case is ignored in arguments to the **−W** option. Each **−W** option has a corresponding long option, as detailed below. Arguments to long options are either joined with the option by an **=** sign, with no intervening spaces, or they may be provided in the next command line argument. Long options may be abbreviated, as long as the abbreviation remains unique.

OPTIONS

Gawk accepts the following options.

−F *fs*
−−field-separator *fs*
Use *fs* for the input field separator (the value of the **FS** predefined variable).

−v *var***=***val*
−−assign *var***=***val*
Assign the value *val*, to the variable *var*, before execution of the program begins. Such variable values are available to the **BEGIN** block of an AWK program.

−f *program-file*
−−file *program-file*
Read the AWK program source from the file *program-file*, instead of from the first command line argument. Multiple **−f** (or **−−file**) options may be used.

−mf=*NNN*
−mr=*NNN*
Set various memory limits to the value *NNN*. The **f** flag sets the maximum number of fields, and the **r** flag sets the maximum record size. These two flags and the **−m** option are from the Bell Labs research version of UNIX *awk*. They are ignored by *gawk*, since *gawk* has no pre-defined limits.

−W traditional
−W compat
−−traditional
−−compat
Run in *compatibility* mode. In compatibility mode, *gawk* behaves identically to UNIX *awk*; none of the GNU-specific extensions are recognized. The use of **−−traditional** is preferred over the other forms of this option. See **GNU EXTENSIONS**, below, for more information.

−W copyleft
−W copyright
−−copyleft
−−copyright
Print the short version of the GNU copyright information message on the error output.

−W help
−W usage
−−help
−−usage
Print a relatively short summary of the available options on the error output. (Per the *GNU Coding Standards*, these options cause an immediate, successful exit.)

−W lint
−−lint Provide warnings about constructs that are dubious or non-portable to other AWK implementations.

−W lint−old
−−lint−old
Provide warnings about constructs that are not portable to the original version of Unix *awk*.

−W posix
−−posix
This turns on *compatibility* mode, with the following additional restrictions:

- **\x** escape sequences are not recognized.
- The synonym **func** for the keyword **function** is not recognized.
- The operators ****** and ****=** cannot be used in place of **^** and **^=**.
- The **fflush()** function is not available.

−W re−interval
−−re−interval
Enable the use of *interval expressions* in regular expression matching (see **Regular Expressions**, below). Interval expressions were not traditionally available in the AWK language. The POSIX standard added them, to make *awk* and *egrep* consistent with each other. However, their use is likely to break old AWK programs, so *gawk* only provides them if they are requested with this option, or when **−−posix** is specified.

−W source *program-text*
−−source *program-text*
Use *program-text* as AWK program source code. This option allows the easy intermixing of library functions (used via the **−f** and **−−file** options) with source code entered on the command line. It is intended primarily for medium to large AWK programs used in shell scripts.

The **−W source=** form of this option uses the rest of the command line argument for *program-text*; no other options to **−W** will be recognized in the same argument.

−W version
−−version
Print version information for this particular copy of *gawk* on the error output. This is useful mainly for knowing if the current copy of *gawk* on your system is up to date with respect to whatever the Free Software Foundation is distributing. This is also useful when reporting bugs. (Per the *GNU Coding Standards*, these options cause an immediate, successful exit.)

−− Signal the end of options. This is useful to allow further arguments to the AWK program itself to start with a ‘‘−’’. This is mainly for consistency with the argument parsing convention used by most other POSIX programs.

In compatibility mode, any other options are flagged as illegal, but are otherwise ignored. In normal operation, as long as program text has been supplied, unknown options are passed on to the AWK program in the

ARGV array for processing. This is particularly useful for running AWK programs via the “#!” executable interpreter mechanism.

AWK PROGRAM EXECUTION

An AWK program consists of a sequence of pattern-action statements and optional function definitions.

> *pattern* **{** *action statements* **}**
> **function** *name***(***parameter list***) {** *statements* **}**

Gawk first reads the program source from the *program-file*(s) if specified, from arguments to **−−source**, or from the first non-option argument on the command line. The **−f** and **−−source** options may be used multiple times on the command line. *Gawk* will read the program text as if all the *program-file*s and command line source texts had been concatenated together. This is useful for building libraries of AWK functions, without having to include them in each new AWK program that uses them. It also provides the ability to mix library functions with command line programs.

The environment variable **AWKPATH** specifies a search path to use when finding source files named with the **−f** option. If this variable does not exist, the default path is **".:/usr/local/share/awk"**. (The actual directory may vary, depending upon how *gawk* was built and installed.) If a file name given to the **−f** option contains a “/” character, no path search is performed.

Gawk executes AWK programs in the following order. First, all variable assignments specified via the **−v** option are performed. Next, *gawk* compiles the program into an internal form. Then, *gawk* executes the code in the **BEGIN** block(s) (if any), and then proceeds to read each file named in the **ARGV** array. If there are no files named on the command line, *gawk* reads the standard input.

If a filename on the command line has the form *var***=***val* it is treated as a variable assignment. The variable *var* will be assigned the value *val*. (This happens after any **BEGIN** block(s) have been run.) Command line variable assignment is most useful for dynamically assigning values to the variables AWK uses to control how input is broken into fields and records. It is also useful for controlling state if multiple passes are needed over a single data file.

If the value of a particular element of **ARGV** is empty (**""**), *gawk* skips over it.

For each record in the input, *gawk* tests to see if it matches any *pattern* in the AWK program. For each pattern that the record matches, the associated *action* is executed. The patterns are tested in the order they occur in the program.

Finally, after all the input is exhausted, *gawk* executes the code in the **END** block(s) (if any).

VARIABLES, RECORDS AND FIELDS

AWK variables are dynamic; they come into existence when they are first used. Their values are either floating-point numbers or strings, or both, depending upon how they are used. AWK also has one dimensional arrays; arrays with multiple dimensions may be simulated. Several pre-defined variables are set as a program runs; these will be described as needed and summarized below.

Records

Normally, records are separated by newline characters. You can control how records are separated by assigning values to the built-in variable **RS**. If **RS** is any single character, that character separates records. Otherwise, **RS** is a regular expression. Text in the input that matches this regular expression will separate the record. However, in compatibility mode, only the first character of its string value is used for separating records. If **RS** is set to the null string, then records are separated by blank lines. When **RS** is set to the null string, the newline character always acts as a field separator, in addition to whatever value **FS** may have.

Fields

As each input record is read, *gawk* splits the record into *fields*, using the value of the **FS** variable as the field separator. If **FS** is a single character, fields are separated by that character. If **FS** is the null string, then each individual character becomes a separate field. Otherwise, **FS** is expected to be a full regular expression. In the special case that **FS** is a single space, fields are separated by runs of spaces and/or tabs. Note that the value of **IGNORECASE** (see below) will also affect how fields are split when **FS** is a regular expression, and how records are separated when **RS** is a regular expression.

If the **FIELDWIDTHS** variable is set to a space separated list of numbers, each field is expected to have fixed width, and *gawk* will split up the record using the specified widths. The value of **FS** is ignored. Assigning a new value to **FS** overrides the use of **FIELDWIDTHS**, and restores the default behavior.

Each field in the input record may be referenced by its position, **$1**, **$2**, and so on. **$0** is the whole record. The value of a field may be assigned to as well. Fields need not be referenced by constants:

```
n = 5
print $n
```

prints the fifth field in the input record. The variable **NF** is set to the total number of fields in the input record.

References to non-existent fields (i.e. fields after **$NF**) produce the null-string. However, assigning to a non-existent field (e.g., **$(NF+2) = 5**) will increase the value of **NF**, create any intervening fields with the null string as their value, and cause the value of **$0** to be recomputed, with the fields being separated by the value of **OFS**. References to negative numbered fields cause a fatal error.

Built-in Variables

Gawk's built-in variables are:

ARGC The number of command line arguments (does not include options to *gawk*, or the program source).

ARGIND The index in **ARGV** of the current file being processed.

ARGV Array of command line arguments. The array is indexed from 0 to **ARGC** − 1. Dynamically changing the contents of **ARGV** can control the files used for data.

CONVFMT The conversion format for numbers, **"%.6g"**, by default.

ENVIRON An array containing the values of the current environment. The array is indexed by the environment variables, each element being the value of that variable (e.g., **ENVIRON["HOME"]** might be **/home/arnold**). Changing this array does not affect the environment seen by programs which *gawk* spawns via redirection or the **system()** function. (This may change in a future version of *gawk*.)

ERRNO If a system error occurs either doing a redirection for **getline**, during a read for **getline**, or during a **close()**, then **ERRNO** will contain a string describing the error.

FIELDWIDTHS A white-space separated list of fieldwidths. When set, *gawk* parses the input into fields of fixed width, instead of using the value of the **FS** variable as the field separator. The fixed field width facility is still experimental; the semantics may change as *gawk* evolves over time.

FILENAME The name of the current input file. If no files are specified on the command line, the value of **FILENAME** is "−". However, **FILENAME** is undefined inside the **BEGIN** block.

FNR The input record number in the current input file.

FS The input field separator, a space by default. See **Fields**, above.

IGNORECASE Controls the case-sensitivity of all regular expression and string operations. If **IGNORECASE** has a non-zero value, then string comparisons and pattern matching in rules, field splitting with **FS**, record separating with **RS**, regular expression matching with **~** and **!~**, and the **gensub()**, **gsub()**, **index()**, **match()**, **split()**, and **sub()** pre-defined functions will all ignore case when doing regular expression operations. Thus, if **IGNORECASE** is not equal to zero, **/aB/** matches all of the strings **"ab"**, **"aB"**, **"Ab"**, and **"AB"**. As with all AWK variables, the initial value of **IGNORECASE** is zero, so all regular expression and string operations are normally case-sensitive. Under Unix, the full ISO 8859-1 Latin-1 character set is used when ignoring case. **NOTE:** In versions of *gawk* prior to 3.0, **IGNORECASE** only affected regular expression operations. It now affects string comparisons as well.

NF The number of fields in the current input record.

NR The total number of input records seen so far.

OFMT The output format for numbers, **"%.6g"**, by default.

OFS The output field separator, a space by default.

ORS The output record separator, by default a newline.

RS The input record separator, by default a newline.

RT The record terminator. *Gawk* sets **RT** to the input text that matched the character or regular expression specified by **RS**.

RSTART The index of the first character matched by **match**(); 0 if no match.

RLENGTH The length of the string matched by **match**(); −1 if no match.

SUBSEP The character used to separate multiple subscripts in array elements, by default **"\034"**.

Arrays

Arrays are subscripted with an expression between square brackets ([and]). If the expression is an expression list (*expr*, *expr* ...) then the array subscript is a string consisting of the concatenation of the (string) value of each expression, separated by the value of the **SUBSEP** variable. This facility is used to simulate multiply dimensioned arrays. For example:

```
i = "A"; j = "B"; k = "C"
x[i, j, k] = "hello, world\n"
```

assigns the string **"hello, world\n"** to the element of the array **x** which is indexed by the string **"A\034B\034C"**. All arrays in AWK are associative, i.e. indexed by string values.

The special operator **in** may be used in an **if** or **while** statement to see if an array has an index consisting of a particular value.

```
if (val in array)
        print array[val]
```

If the array has multiple subscripts, use **(i, j) in array**.

The **in** construct may also be used in a **for** loop to iterate over all the elements of an array.

An element may be deleted from an array using the **delete** statement. The **delete** statement may also be used to delete the entire contents of an array, just by specifying the array name without a subscript.

Variable Typing And Conversion

Variables and fields may be (floating point) numbers, or strings, or both. How the value of a variable is interpreted depends upon its context. If used in a numeric expression, it will be treated as a number, if used as a string it will be treated as a string.

To force a variable to be treated as a number, add 0 to it; to force it to be treated as a string, concatenate it with the null string.

When a string must be converted to a number, the conversion is accomplished using *atof*(3). A number is converted to a string by using the value of **CONVFMT** as a format string for *sprintf*(3), with the numeric value of the variable as the argument. However, even though all numbers in AWK are floating-point, integral values are *always* converted as integers. Thus, given

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

the variable **b** has a string value of **"12"** and not **"12.00"**.

Gawk performs comparisons as follows: If two variables are numeric, they are compared numerically. If one value is numeric and the other has a string value that is a ‘‘numeric string,’’ then comparisons are also done numerically. Otherwise, the numeric value is converted to a string and a string comparison is performed. Two strings are compared, of course, as strings. According to the POSIX standard, even if two strings are numeric strings, a numeric comparison is performed. However, this is clearly incorrect, and *gawk* does not do this.

Note that string constants, such as **"57"**, are *not* numeric strings, they are string constants. The idea of ‘‘numeric string’’ only applies to fields, **getline** input, **FILENAME**, **ARGV** elements, **ENVIRON** elements and the elements of an array created by **split()** that are numeric strings. The basic idea is that *user input*, and only user input, that looks numeric, should be treated that way.

Uninitialized variables have the numeric value 0 and the string value "" (the null, or empty, string).

PATTERNS AND ACTIONS

AWK is a line oriented language. The pattern comes first, and then the action. Action statements are enclosed in **{** and **}**. Either the pattern may be missing, or the action may be missing, but, of course, not both. If the pattern is missing, the action will be executed for every single record of input. A missing action is equivalent to

{ print }

which prints the entire record.

Comments begin with the ‘‘#’’ character, and continue until the end of the line. Blank lines may be used to separate statements. Normally, a statement ends with a newline, however, this is not the case for lines ending in a ‘‘,’’, **{**, **?**, **:**, **&&**, or **||**. Lines ending in **do** or **else** also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a ‘‘\’’, in which case the newline will be ignored.

Multiple statements may be put on one line by separating them with a ‘‘;’’. This applies to both the statements within the action part of a pattern-action pair (the usual case), and to the pattern-action statements themselves.

Patterns

AWK patterns may be one of the following:

BEGIN
END
/*regular expression***/**
relational expression
pattern **&&** *pattern*
pattern **||** *pattern*
pattern **?** *pattern* **:** *pattern*
(*pattern***)**
! *pattern*
*pattern1***,** *pattern2*

BEGIN and **END** are two special kinds of patterns which are not tested against the input. The action parts of all **BEGIN** patterns are merged as if all the statements had been written in a single **BEGIN** block. They are executed before any of the input is read. Similarly, all the **END** blocks are merged, and executed when all the input is exhausted (or when an **exit** statement is executed). **BEGIN** and **END** patterns cannot be combined with other patterns in pattern expressions. **BEGIN** and **END** patterns cannot have missing action parts.

For **/***regular expression***/** patterns, the associated statement is executed for each input record that matches the regular expression. Regular expressions are the same as those in *egrep*(1), and are summarized below.

A *relational expression* may use any of the operators defined below in the section on actions. These generally test whether certain fields match certain regular expressions.

The **&&**, **||**, and **!** operators are logical AND, logical OR, and logical NOT, respectively, as in C. They do short-circuit evaluation, also as in C, and are used for combining more primitive pattern expressions. As in most languages, parentheses may be used to change the order of evaluation.

The **?:** operator is like the same operator in C. If the first pattern is true then the pattern used for testing is the second pattern, otherwise it is the third. Only one of the second and third patterns is evaluated.

The *pattern1*, *pattern2* form of an expression is called a *range pattern*. It matches all input records starting with a record that matches *pattern1*, and continuing until a record that matches *pattern2*, inclusive. It does not combine with any other sort of pattern expression.

Regular Expressions

Regular expressions are the extended kind found in *egrep*. They are composed of characters as follows:

c matches the non-metacharacter *c*.

c matches the literal character *c*.

. matches any character *including* newline.

^ matches the beginning of a string.

$ matches the end of a string.

[*abc...***]** character list, matches any of the characters *abc....*

[^*abc...***]** negated character list, matches any character except *abc...* and newline.

*r1***|***r2* alternation: matches either *r1* or *r2*.

r1r2 concatenation: matches *r1*, and then *r2*.

*r***+** matches one or more *r*'s.

*r****** matches zero or more *r*'s.

*r***?** matches zero or one *r*'s.

(*r***)** grouping: matches *r*.

*r***{***n***}**
*r***{***n***,}**
*r***{***n***,***m***}** One or two numbers inside braces denote an *interval expression*. If there is one number in the braces, the preceding regexp *r* is repeated *n* times. If there are two numbers separated by a comma, *r* is repeated *n* to *m* times. If there is one number followed by a comma, then *r* is repeated at least *n* times.

Interval expressions are only available if either **−−posix** or **−−re−interval** is specified on the command line.

\y matches the empty string at either the beginning or the end of a word.

\B matches the empty string within a word.

\< matches the empty string at the beginning of a word.

\> matches the empty string at the end of a word.

\w matches any word-constituent character (letter, digit, or underscore).

\W matches any character that is not word-constituent.

\` matches the empty string at the beginning of a buffer (string).

\' matches the empty string at the end of a buffer.

The escape sequences that are valid in string constants (see below) are also legal in regular expressions.

Character classes are a new feature introduced in the POSIX standard. A character class is a special notation for describing lists of characters that have a specific attribute, but where the actual characters themselves can vary from country to country and/or from character set to character set. For example, the notion of what is an alphabetic character differs in the USA and in France.

A character class is only valid in a regexp *inside* the brackets of a character list. Character classes consist of **[:**, a keyword denoting the class, and **:]**. Here are the character classes defined by the POSIX standard.

[:alnum:]
Alphanumeric characters.

[:alpha:]
Alphabetic characters.

[:blank:]
Space or tab characters.

[:cntrl:]
Control characters.

[:digit:]
Numeric characters.

[:graph:]
Characters that are both printable and visible. (A space is printable, but not visible, while an **a** is both.)

[:lower:]
Lower-case alphabetic characters.

[:print:]
Printable characters (characters that are not control characters.)

[:punct:]
Punctuation characters (characters that are not letter, digits, control characters, or space characters).

[:space:]
Space characters (such as space, tab, and formfeed, to name a few).

[:upper:]
Upper-case alphabetic characters.

[:xdigit:]
Characters that are hexadecimal digits.

For example, before the POSIX standard, to match alphanumeric characters, you would have had to write **/[A−Za−z0−9]/**. If your character set had other alphabetic characters in it, this would not match them. With the POSIX character classes, you can write **/[[:alnum:]]/**, and this will match *all* the alphabetic and numeric characters in your character set.

Two additional special sequences can appear in character lists. These apply to non-ASCII character sets, which can have single symbols (called *collating elements*) that are represented with more than one character, as well as several characters that are equivalent for *collating*, or sorting, purposes. (E.g., in French, a plain ‘‘e’’ and a grave-accented e` are equivalent.)

Collating Symbols
A collating symbols is a multi-character collating element enclosed in **[.** and **.]**. For example, if **ch** is a collating element, then **[[.ch.]]** is a regexp that matches this collating element, while **[ch]** is a regexp that matches either **c** or **h**.

Equivalence Classes
An equivalence class is a list of equivalent characters enclosed in **[=** and **=]**. Thus, **[[=ee`=]]** is regexp that matches either **e** or **e`** .

These features are very valuable in non-English speaking locales. The library functions that *gawk* uses for regular expression matching currently only recognize POSIX character classes; they do not recognize collating symbols or equivalence classes.

The **\y**, **\B**, **\<**, **\>**, **\w**, **\W**, **\‘**, and **\’** operators are specific to *gawk*; they are extensions based on facilities in the GNU regexp libraries.

The various command line options control how *gawk* interprets characters in regexps.

No options
: In the default case, *gawk* provide all the facilities of POSIX regexps and the GNU regexp operators described above. However, interval expressions are not supported.

−−posix
: Only POSIX regexps are supported, the GNU operators are not special. (E.g., **\w** matches a literal **w**). Interval expressions are allowed.

−−traditional
: Traditional Unix *awk* regexps are matched. The GNU operators are not special, interval expressions are not available, and neither are the POSIX character classes (**[[:alnum:]]** and so on). Characters described by octal and hexadecimal escape sequences are treated literally, even if they represent regexp metacharacters.

−−re−interval
: Allow interval expressions in regexps, even if **−−traditional** has been provided.

Actions

Action statements are enclosed in braces, **{** and **}**. Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and input/output statements available are patterned after those in C.

Operators

The operators in AWK, in order of decreasing precedence, are

(...)	Grouping
$	Field reference.
++ −−	Increment and decrement, both prefix and postfix.
^	Exponentiation (****** may also be used, and ****=** for the assignment operator).
+ − !	Unary plus, unary minus, and logical negation.
*** / %**	Multiplication, division, and modulus.
+ −	Addition and subtraction.
space	String concatenation.
< > **<= >=** **!= ==**	The regular relational operators.
~ !~	Regular expression match, negated match. **NOTE:** Do not use a constant regular expression (**/foo/**) on the left-hand side of a **~** or **!~**. Only use one on the right-hand side. The expression **/foo/ ~** *exp* has the same meaning as **(($0 ~ /foo/) ~** *exp*). This is usually *not* what was intended.
in	Array membership.
&&	Logical AND.
||	Logical OR.
?:	The C conditional expression. This has the form *expr1* **?** *expr2* **:** *expr3*. If *expr1* is true, the value of the expression is *expr2*, otherwise it is *expr3*. Only one of *expr2* and *expr3* is evaluated.
= += −= ***= /= %= ^=**	Assignment. Both absolute assignment (*var* **=** *value*) and operator-assignment (the other forms) are supported.

Control Statements

The control statements are as follows:

if (*condition***)** *statement* [**else** *statement*]
while (*condition***)** *statement*
do *statement* **while (***condition***)**
for (*expr1***;** *expr2***;** *expr3***)** *statement*
for (*var* **in** *array***)** *statement*
break
continue
delete *array***[***index***]**
delete *array*
exit [*expression*]
{ *statements* **}**

I/O Statements

The input/output statements are as follows:

close(*file***)** Close file (or pipe, see below).

getline Set **$0** from next input record; set **NF**, **NR**, **FNR**.

getline <*file* Set **$0** from next record of *file*; set **NF**.

getline *var* Set *var* from next input record; set **NF**, **FNR**.

getline *var* **<***file* Set *var* from next record of *file*.

next Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, the **END** block(s), if any, are executed.

nextfile Stop processing the current input file. The next input record read comes from the next input file. **FILENAME** and **ARGIND** are updated, **FNR** is reset to 1, and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, the **END** block(s), if any, are executed. **NOTE:** Earlier versions of gawk used **next file**, as two words. While this usage is still recognized, it generates a warning message and will eventually be removed.

print Prints the current record. The output record is terminated with the value of the **ORS** variable.

print *expr-list* Prints expressions. Each expression is separated by the value of the **OFS** variable. The output record is terminated with the value of the **ORS** variable.

print *expr-list* **>***file*
Prints expressions on *file*. Each expression is separated by the value of the **OFS** variable. The output record is terminated with the value of the **ORS** variable.

printf *fmt, expr-list* Format and print.

printf *fmt, expr-list* **>***file*
Format and print on *file*.

system(*cmd-line***)** Execute the command *cmd-line*, and return the exit status. (This may not be available on non-POSIX systems.)

fflush([*file*]**)** Flush any buffers associated with the open output file or pipe *file*. If *file* is missing, then standard output is flushed. If *file* is the null string, then all open output files and pipes have their buffers flushed.

Other input/output redirections are also allowed. For **print** and **printf**, **>>***file* appends output to the *file*, while **|** *command* writes on a pipe. In a similar fashion, *command* **| getline** pipes into **getline**. The **getline** command will return 0 on end of file, and −1 on an error.

The *printf* Statement

The AWK versions of the **printf** statement and **sprintf()** function (see below) accept the following conversion specification formats:

%c An ASCII character. If the argument used for **%c** is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

%d
%i A decimal number (the integer part).

%e
%E A floating point number of the form **[−]d.dddddde[+−]dd**. The **%E** format uses **E** instead of **e**.

%f A floating point number of the form **[−]ddd.dddddd**.

%g
%G Use **%e** or **%f** conversion, whichever is shorter, with nonsignificant zeros suppressed. The **%G** format uses **%E** instead of **%e**.

%o An unsigned octal number (again, an integer).

%s A character string.

%x
%X An unsigned hexadecimal number (an integer). **%X** format uses **ABCDEF** instead of **abcdef**.

%% A single **%** character; no argument is converted.

There are optional, additional parameters that may lie between the **%** and the control letter:

− The expression should be left-justified within its field.

space For numeric conversions, prefix positive values with a space, and negative values with a minus sign.

+ The plus sign, used before the width modifier (see below), says to always supply a sign for numeric conversions, even if the data to be formatted is positive. The **+** overrides the space modifier.

Use an ‘‘alternate form’’ for certain control letters. For **%o**, supply a leading zero. For **%x**, and **%X**, supply a leading **0x** or **0X** for a nonzero result. For **%e**, **%E**, and **%f**, the result will always contain a decimal point. For **%g**, and **%G**, trailing zeros are not removed from the result.

0 A leading **0** (zero) acts as a flag, that indicates output should be padded with zeroes instead of spaces. This applies even to non-numeric output formats. This flag only has an effect when the field width is wider than the value to be printed.

width The field should be padded to this width. The field is normally padded with spaces. If the **0** flag has been used, it is padded with zeroes.

.*prec* A number that specifies the precision to use when printing. For the **%e**, **%E**, and **%f** formats, this specifies the number of digits you want printed to the right of the decimal point. For the **%g**, and **%G** formats, it specifies the maximum number of significant digits. For the **%d**, **%o**, **%i**, **%u**, **%x**, and **%X** formats, it specifies the minimum number of digits to print. For a string, it specifies the maximum number of characters from the string that should be printed.

The dynamic *width* and *prec* capabilities of the ANSI C **printf()** routines are supported. A ***** in place of either the **width** or **prec** specifications will cause their values to be taken from the argument list to **printf** or **sprintf()**.

Special File Names

When doing I/O redirection from either **print** or **printf** into a file, or via **getline** from a file, *gawk* recognizes certain special filenames internally. These filenames allow access to open file descriptors inherited from *gawk*’s parent process (usually the shell). Other special filenames provide access to information about the running **gawk** process. The filenames are:

/dev/pid Reading this file returns the process ID of the current process, in decimal, terminated with a newline.

/dev/ppid Reading this file returns the parent process ID of the current process, in decimal, terminated with a newline.

/dev/pgrpid
Reading this file returns the process group ID of the current process, in decimal, terminated with a newline.

/dev/user Reading this file returns a single record terminated with a newline. The fields are separated with spaces. **$1** is the value of the *getuid*(2) system call, **$2** is the value of the *geteuid*(2) system call, **$3** is the value of the *getgid*(2) system call, and **$4** is the value of the *getegid*(2) system call. If there are any additional fields, they are the group IDs returned by *getgroups*(2). Multiple groups may not be supported on all systems.

/dev/stdin The standard input.

/dev/stdout The standard output.

/dev/stderr The standard error output.

/dev/fd/*n* The file associated with the open file descriptor *n*.

These are particularly useful for error messages. For example:

```
print "You blew it!" > "/dev/stderr"
```

whereas you would otherwise have to use

```
print "You blew it!" | "cat 1>&2"
```

These file names may also be used on the command line to name data files.

Numeric Functions

AWK has the following pre-defined arithmetic functions:

atan2(*y***,** *x***)** returns the arctangent of *y/x* in radians.

cos(*expr***)** returns the cosine in radians.

exp(*expr***)** the exponential function.

int(*expr***)** truncates to integer.

log(*expr***)** the natural logarithm function.

rand() returns a random number between 0 and 1.

sin(*expr***)** returns the sine in radians.

sqrt(*expr***)** the square root function.

srand([*expr*]**)** uses *expr* as a new seed for the random number generator. If no *expr* is provided, the time of day will be used. The return value is the previous seed for the random number generator.

String Functions

Gawk has the following pre-defined string functions:

gensub(*r***,** *s***,** *h* [**,** *t*]**)** search the target string *t* for matches of the regular expression *r*. If *h* is a string beginning with **g** or **G**, then replace all matches of *r* with *s*. Otherwise, *h* is a number indicating which match of *r* to replace. If no *t* is supplied, **$0** is used instead. Within the replacement text *s*, the sequence *****n*, where *n* is a digit from 1 to 9, may be used to indicate just the text that matched the *n*'th parenthesized subexpression. The sequence **\0** represents the entire matched text, as does the character **&**. Unlike **sub()** and **gsub()**, the modified string is returned as the result of the function, and the original target string is *not* changed.

gsub(*r*, *s* [, *t*]) for each substring matching the regular expression *r* in the string *t*, substitute the string *s*, and return the number of substitutions. If *t* is not supplied, use **$0**. An **&** in the replacement text is replaced with the text that was actually matched. Use **\&** to get a literal **&**. See *AWK Language Programming* for a fuller discussion of the rules for **&'s** and backslashes in the replacement text of **sub()**, **gsub()**, and **gensub()**.

index(*s*, *t*) returns the index of the string *t* in the string *s*, or 0 if *t* is not present.

length([*s*]) returns the length of the string *s*, or the length of **$0** if *s* is not supplied.

match(*s*, *r*) returns the position in *s* where the regular expression *r* occurs, or 0 if *r* is not present, and sets the values of **RSTART** and **RLENGTH**.

split(*s*, *a* [, *r*]) splits the string *s* into the array *a* on the regular expression *r*, and returns the number of fields. If *r* is omitted, **FS** is used instead. The array *a* is cleared first. Splitting behaves identically to field splitting, described above.

sprintf(*fmt*, *expr-list*)
prints *expr-list* according to *fmt*, and returns the resulting string.

sub(*r*, *s* [, *t*]) just like **gsub**(), but only the first matching substring is replaced.

substr(*s*, *i* [, *n*]) returns the at most *n*-character substring of *s* starting at *i*. If *n* is omitted, the rest of *s* is used.

tolower(*str*) returns a copy of the string *str*, with all the upper-case characters in *str* translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

toupper(*str*) returns a copy of the string *str*, with all the lower-case characters in *str* translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

Time Functions

Since one of the primary uses of AWK programs is processing log files that contain time stamp information, *gawk* provides the following two functions for obtaining time stamps and formatting them.

systime() returns the current time of day as the number of seconds since the Epoch (Midnight UTC, January 1, 1970 on POSIX systems).

strftime([*format* [, *timestamp*]])
formats *timestamp* according to the specification in *format.* The *timestamp* should be of the same form as returned by **systime**(). If *timestamp* is missing, the current time of day is used. If *format* is missing, a default format equivalent to the output of *date*(1) will be used. See the specification for the **strftime()** function in ANSI C for the format conversions that are guaranteed to be available. A public-domain version of *strftime*(3) and a man page for it come with *gawk*; if that version was used to build *gawk*, then all of the conversions described in that man page are available to *gawk.*

String Constants

String constants in AWK are sequences of characters enclosed between double quotes ("). Within strings, certain *escape sequences* are recognized, as in C. These are:

**** A literal backslash.

\a The "alert" character; usually the ASCII BEL character.

\b backspace.

\f form-feed.

\n newline.

\r carriage return.

\t horizontal tab.

\v vertical tab.

\x*hex digits*
The character represented by the string of hexadecimal digits following the **\x**. As in ANSI C, all following hexadecimal digits are considered part of the escape sequence. (This feature should tell us something about language design by committee.) E.g., **"\x1B"** is the ASCII ESC (escape) character.

ddd The character represented by the 1-, 2-, or 3-digit sequence of octal digits. E.g. **"\033"** is the ASCII ESC (escape) character.

c The literal character *c*.

The escape sequences may also be used inside constant regular expressions (e.g., **/[\t\f\n\r\v]/** matches whitespace characters).

In compatibility mode, the characters represented by octal and hexadecimal escape sequences are treated literally when used in regexp constants. Thus, **/a\52b/** is equivalent to **/a*b/**.

FUNCTIONS

Functions in AWK are defined as follows:

function *name*(*parameter list*) **{** *statements* **}**

Functions are executed when they are called from within expressions in either patterns or actions. Actual parameters supplied in the function call are used to instantiate the formal parameters declared in the function. Arrays are passed by reference, other variables are passed by value.

Since functions were not originally part of the AWK language, the provision for local variables is rather clumsy: They are declared as extra parameters in the parameter list. The convention is to separate local variables from real parameters by extra spaces in the parameter list. For example:

```
function  f(p, q,     a, b)    # a & b are local
{
          .....
}

/abc/     { ... ; f(1, 2) ; ... }
```

The left parenthesis in a function call is required to immediately follow the function name, without any intervening white space. This is to avoid a syntactic ambiguity with the concatenation operator. This restriction does not apply to the built-in functions listed above.

Functions may call each other and may be recursive. Function parameters used as local variables are initialized to the null string and the number zero upon function invocation.

If **−−lint** has been provided, *gawk* will warn about calls to undefined functions at parse time, instead of at run time. Calling an undefined function at run time is a fatal error.

The word **func** may be used in place of **function**.

EXAMPLES

Print and sort the login names of all users:

```
BEGIN { FS = ":" }
      { print $1 | "sort" }
```

Count lines in a file:

```
      { nlines++ }
END   { print nlines }
```

Precede each line by its number in the file:

{ print FNR, $0 }

Concatenate and line number (a variation on a theme):

{ print NR, $0 }

SEE ALSO

egrep(1), *getpid*(2), *getppid*(2), *getpgrp*(2), *getuid*(2), *geteuid*(2), *getgid*(2), *getegid*(2), *getgroups*(2)

The AWK Programming Language, Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, Addison-Wesley, 1988. ISBN 0-201-07981-X.

AWK Language Programming, Edition 1.0, published by the Free Software Foundation, 1995.

POSIX COMPATIBILITY

A primary goal for *gawk* is compatibility with the POSIX standard, as well as with the latest version of UNIX *awk*. To this end, *gawk* incorporates the following user visible features which are not described in the AWK book, but are part of the Bell Labs version of *awk*, and are in the POSIX standard.

The **−v** option for assigning variables before program execution starts is new. The book indicates that command line variable assignment happens when *awk* would otherwise open the argument as a file, which is after the **BEGIN** block is executed. However, in earlier implementations, when such an assignment appeared before any file names, the assignment would happen *before* the **BEGIN** block was run. Applications came to depend on this "feature." When *awk* was changed to match its documentation, this option was added to accommodate applications that depended upon the old behavior. (This feature was agreed upon by both the AT&T and GNU developers.)

The **−W** option for implementation specific features is from the POSIX standard.

When processing arguments, *gawk* uses the special option "**−−**" to signal the end of arguments. In compatibility mode, it will warn about, but otherwise ignore, undefined options. In normal operation, such arguments are passed on to the AWK program for it to process.

The AWK book does not define the return value of **srand()**. The POSIX standard has it return the seed it was using, to allow keeping track of random number sequences. Therefore **srand()** in *gawk* also returns its current seed.

Other new features are: The use of multiple **−f** options (from MKS *awk*); the **ENVIRON** array; the **\a**, and **\v** escape sequences (done originally in *gawk* and fed back into AT&T's); the **tolower()** and **toupper()** built-in functions (from AT&T); and the ANSI C conversion specifications in **printf** (done first in AT&T's version).

GNU EXTENSIONS

Gawk has a number of extensions to POSIX *awk*. They are described in this section. All the extensions described here can be disabled by invoking *gawk* with the **−−traditional** option.

The following features of *gawk* are not available in POSIX *awk*.

- The **\x** escape sequence. (Disabled with **−−posix**.)
- The **fflush()** function. (Disabled with **−−posix**.)
- The **systime(), strftime(),** and **gensub()** functions.
- The special file names available for I/O redirection are not recognized.
- The **ARGIND**, **ERRNO**, and **RT** variables are not special.
- The **IGNORECASE** variable and its side-effects are not available.
- The **FIELDWIDTHS** variable and fixed-width field splitting.
- The use of **RS** as a regular expression.

- The ability to split out individual characters using the null string as the value of **FS**, and as the third argument to **split()**.
- No path search is performed for files named via the **−f** option. Therefore the **AWKPATH** environment variable is not special.
- The use of **nextfile** to abandon processing of the current input file.
- The use of **delete** *array* to delete the entire contents of an array.

The AWK book does not define the return value of the **close()** function. *Gawk*'s **close()** returns the value from *fclose*(3), or *pclose*(3), when closing a file or pipe, respectively.

When *gawk* is invoked with the **−−traditional** option, if the *fs* argument to the **−F** option is ‘‘t’’, then **FS** will be set to the tab character. Since this is a rather ugly special case, it is not the default behavior. This behavior also does not occur if **−−posix** has been specified.

HISTORICAL FEATURES

There are two features of historical AWK implementations that *gawk* supports. First, it is possible to call the **length()** built-in function not only with no argument, but even without parentheses! Thus,

```
a = length     # Holy Algol 60, Batman!
```

is the same as either of

```
a = length()
a = length($0)
```

This feature is marked as ‘‘deprecated’’ in the POSIX standard, and *gawk* will issue a warning about its use if **−−lint** is specified on the command line.

The other feature is the use of either the **continue** or the **break** statements outside the body of a **while**, **for**, or **do** loop. Traditional AWK implementations have treated such usage as equivalent to the **next** statement. *Gawk* will support this usage if **−−traditional** has been specified.

ENVIRONMENT VARIABLES

If **POSIXLY_CORRECT** exists in the environment, then *gawk* behaves exactly as if **−−posix** had been specified on the command line. If **−−lint** has been specified, *gawk* will issue a warning message to this effect.

The **AWKPATH** environment variable can be used to provide a list of directories that *gawk* will search when looking for files named via the **−f** and **−−file** options.

BUGS

The **−F** option is not necessary given the command line variable assignment feature; it remains only for backwards compatibility.

If your system actually has support for **/dev/fd** and the associated **/dev/stdin**, **/dev/stdout**, and **/dev/stderr** files, you may get different output from *gawk* than you would get on a system without those files. When *gawk* interprets these files internally, it synchronizes output to the standard output with output to **/dev/stdout**, while on a system with those files, the output is actually to different open files. Caveat Emptor.

Syntactically invalid single character programs tend to overflow the parse stack, generating a rather unhelpful message. Such programs are surprisingly difficult to diagnose in the completely general case, and the effort to do so really is not worth it.

The word ‘‘GNU’’ is incorrectly capitalized in at least one file in the source code.

VERSION INFORMATION

This man page documents *gawk*, version 3.0.

AUTHORS

The original version of UNIX *awk* was designed and implemented by Alfred Aho, Peter Weinberger, and Brian Kernighan of AT&T Bell Labs. Brian Kernighan continues to maintain and enhance it.

Paul Rubin and Jay Fenlason, of the Free Software Foundation, wrote *gawk*, to be compatible with the original version of *awk* distributed in Seventh Edition UNIX. John Woods contributed a number of bug fixes. David Trueman, with contributions from Arnold Robbins, made *gawk* compatible with the new version of UNIX *awk*. Arnold Robbins is the current maintainer.

The initial DOS port was done by Conrad Kwok and Scott Garfinkle. Scott Deifik is the current DOS maintainer. Pat Rankin did the port to VMS, and Michal Jaegermann did the port to the Atari ST. The port to OS/2 was done by Kai Uwe Rommel, with contributions and help from Darrel Hankerson. Fred Fish supplied support for the Amiga.

BUG REPORTS

If you find a bug in *gawk*, please send electronic mail to **bug-gnu-utils@prep.ai.mit.edu**, *with* a carbon copy to **arnold@gnu.ai.mit.edu**. Please include your operating system and its revision, the version of *gawk*, what C compiler you used to compile it, and a test program and data that are as small as possible for reproducing the problem.

Before sending a bug report, please do two things. First, verify that you have the latest version of *gawk*. Many bugs (usually subtle ones) are fixed at each release, and if yours is out of date, the problem may already have been solved. Second, please read this man page and the reference manual carefully to be sure that what you think is a bug really is, instead of just a quirk in the language.

Whatever you do, do **NOT** post a bug report in **comp.lang.awk**. While the *gawk* developers occasionally read this newsgroup, posting bug reports there is an unreliable way to report bugs. Instead, please use the electronic mail addresses given above.

ACKNOWLEDGEMENTS

Brian Kernighan of Bell Labs provided valuable assistance during testing and debugging. We thank him.