GAWK

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 . . .

pgawk [POSIX or GNU style options] **–f** *program-file* [**––**] file . . .
pgawk [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 1003.1 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 Laboratories *awk* extensions, and a number of GNU-specific extensions.

Pgawk is the profiling version of *gawk*. It is identical in every way to *gawk*, except that programs run more slowly, and it automatically produces an execution profile in the file **awkprof.out** when done. See the **––profile** option, below.

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 traditional POSIX one letter options, or 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 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, listed by frequency.

–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 an earlier version of the Bell Laboratories research version of UNIX *awk*. They are ignored by *gawk*, since *gawk* has no pre-defined limits.

–W compat
–W traditional

−−compat
−−traditional
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 standard output and exit successfully.

−W dump-variables[=*file*]
−−dump-variables[=*file*]
Print a sorted list of global variables, their types and final values to *file*. If no *file* is provided, *gawk* uses a file named **awkvars.out** in the current directory.

Having a list of all the global variables is a good way to look for typographical errors in your programs. You would also use this option if you have a large program with a lot of functions, and you want to be sure that your functions don't inadvertently use global variables that you meant to be local. (This is a particularly easy mistake to make with simple variable names like **i**, **j**, and so on.)

−W exec *file*
−−exec *file*
Similar to **−f**, however, this is option is the last one processed. This should be used with **#!** scripts, particularly for CGI applications, to avoid passing in options or source code (!) on the command line from a URL. This option disables command-line variable assignments.

−W gen−po
−−gen−po
Scan and parse the AWK program, and generate a GNU **.po** format file on standard output with entries for all localizable strings in the program. The program itself is not executed. See the GNU *gettext* distribution for more information on **.po** files.

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

−W lint[=*value*]
−−lint[=*value*]
Provide warnings about constructs that are dubious or non-portable to other AWK implementations. With an optional argument of **fatal**, lint warnings become fatal errors. This may be drastic, but its use will certainly encourage the development of cleaner AWK programs. With an optional argument of **invalid**, only warnings about things that are actually invalid are issued. (This is not fully implemented yet.)

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

−W non−decimal−data
−−non−decimal−data
Recognize octal and hexadecimal values in input data. *Use this option with great caution!*

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

- **\x** escape sequences are not recognized.

- Only space and tab act as field separators when **FS** is set to a single space, newline does not.
- You cannot continue lines after **?** and **:**.
- 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 profile[=*prof_file*]
−−profile[=*prof_file*]
Send profiling data to *prof_file*. The default is **awkprof.out**. When run with *gawk*, the profile is just a "pretty printed" version of the program. When run with *pgawk*, the profile contains execution counts of each statement in the program in the left margin and function call counts for each user-defined function.

−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.

−W use−lc−numeric
−−use−lc−numeric
This forces *gawk* to use the locale's decimal point character when parsing input data. Although the POSIX standard requires this behavior, and *gawk* does so when **−−posix** is in effect, the default is to follow traditional behavior and use a period as the decimal point, even in locales where the period is not the decimal point character. This option overrides the default behavior, without the full draconian strictness of the **−−posix** option.

−W version
−−version
Print version information for this particular copy of *gawk* on the standard 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 provides consistency with the argument parsing convention used by most other POSIX programs.

In compatibility mode, any other options are flagged as invalid, 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* reads 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 are 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 separates 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 and/or newlines. (But see the section **POSIX COMPATIBILITY**, below). **NOTE**: The value of **IGNORECASE** (see below) also affects 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* splits 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. 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**) increases the value of **NF**, creates any intervening fields with the null string as their value, and causes 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. Decrementing **NF** causes the values of fields past the new value to be lost, and the value of **$0** to be recomputed, with the fields being separated by the value of **OFS**.

Assigning a value to an existing field causes the whole record to be rebuilt when **$0** is referenced. Similarly, assigning a value to **$0** causes the record to be resplit, creating new values for the fields.

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.
BINMODE	On non-POSIX systems, specifies use of “binary” mode for all file I/O. Numeric values of 1, 2, or 3, specify that input files, output files, or all files, respectively, should use binary I/O. String values of **"r"**, or **"w"** specify that input files, or output files, respectively, should use binary I/O. String values of **"rw"** or **"wr"** specify that all files should use binary I/O. Any other string value is treated as **"rw"**, but generates a warning message.
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.
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. The value is subject to translation in non-English locales.
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.
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 (unless set by **getline**).
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**() built-in functions all ignore case when doing regular expression operations. **NOTE**: Array subscripting is *not* affected. However, the **asort()** and **asorti()** functions are affected.
	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. As of *gawk* 3.1.4, the case equivalencies are fully locale-aware, based on the C **<ctype.h>** facilities such as **isalpha**(), and **toupper**().
LINT	Provides dynamic control of the **−−lint** option from within an AWK program. When true, *gawk* prints lint warnings. When false, it does not. When assigned the string value **"fatal"**, lint warnings become fatal errors, exactly like **−−lint=fatal**. Any other true value just prints warnings.
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.
PROCINFO	The elements of this array provide access to information about the running AWK program. On some systems, there may be elements in the array, **"group1"** through **"group***n***"** for some *n*, which is the number of supplementary groups that the process has. Use the **in** operator to test for these elements. The following elements are guaranteed to be available:
	PROCINFO["egid"] the value of the *getegid*(2) system call.
	PROCINFO["euid"] the value of the *geteuid*(2) system call.
	PROCINFO["FS"] **"FS"** if field splitting with **FS** is in effect, or **"FIELDWIDTHS"** if field splitting with **FIELDWIDTHS** is in effect.

PROCINFO["gid"]	the value of the *getgid*(2) system call.
PROCINFO["pgrpid"]	the process group ID of the current process.
PROCINFO["pid"]	the process ID of the current process.
PROCINFO["ppid"]	the parent process ID of the current process.
PROCINFO["uid"]	the value of the *getuid*(2) system call.
PROCINFO["version"]	The version of *gawk*. This is available from version 3.1.4 and later.

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. (This implies that character indices start at one.)
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"**.
TEXTDOMAIN	The text domain of the AWK program; used to find the localized translations for the program's strings.

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 to test 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 *strtod*(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"**.

When operating in POSIX mode (such as with the **−−posix** command line option), beware that locale settings may interfere with the way decimal numbers are treated: the decimal separator of the numbers you are feeding to *gawk* must conform to what your locale would expect, be it a comma (,) or a period (.).

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.

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).

Octal and Hexadecimal Constants

Starting with version 3.1 of *gawk* , you may use C-style octal and hexadecimal constants in your AWK program source code. For example, the octal value **011** is equal to decimal **9**, and the hexadecimal value **0x11** is equal to decimal 17.

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 regular expression constants. Thus, **/a\52b/** is equivalent to **/a\*b/**.

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 is 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....*
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 regular expression *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 valid in regular expressions.

Character classes are a 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 regular expression *inside* the brackets of a character list. Character classes consist of **[:**, a keyword denoting the class, and **:]**. The character classes defined by the POSIX standard are:

[: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, and if your character set collated differently from ASCII, this might not even match the ASCII alphanumeric characters. With the POSIX character classes, you can write **/[[:alnum:]]/**, and this matches the alphabetic and numeric characters in your character set, no matter what it is.

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 "è" are equivalent.)

Collating Symbols

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

Equivalence Classes

An equivalence class is a locale-specific name for a list of characters that are equivalent. The name is enclosed in **[=** and **=]**. For example, the name **e** might be used to represent all of "e," "é," and "è." In this case, **[[=e=]]** is a regular expression that matches any of **e**, **é**, or **è**.

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 regular expression libraries.

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

No options

In the default case, *gawk* provide all the facilities of POSIX regular expressions and the GNU regular expression operators described above. However, interval expressions are not supported.

––posix

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

––traditional

Traditional Unix *awk* regular expressions 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 regular expression metacharacters.

−−re−interval
Allow interval expressions in regular expressions, 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.
| |&	Piped I/O for **getline**, **print**, and **printf**.
< > **<= >=** **!= ==**	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* [**,** *how*]**)** Close file, pipe or co-process. The optional *how* should only be used when closing one end of a two-way pipe to a co-process. It must be a string value, either **"to"** or **"from"**.

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 **NR**, **FNR**.

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

command **| getline** [*var*]
Run *command* piping the output either into **$0** or *var*, as above.

command **|& getline** [*var*]
Run *command* as a co-process piping the output either into **$0** or *var*, as above. Co-processes are a *gawk* extension. (*command* can also be a socket. See the subsection **Special File Names**, below.)

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.

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.

Additional output redirections are allowed for **print** and **printf**.

print ... >> *file*
Appends output to the *file*.

print ... | *command*
Writes on a pipe.

print ... |& *command*
Sends data to a co-process or socket. (See also the subsection **Special File Names**, below.)

The **getline** command returns 0 on end of file and −1 on an error. Upon an error, **ERRNO** contains a string describing the problem.

NOTE: If using a pipe, co-process, or socket to **getline**, or from **print** or **printf** within a loop, you *must* use **close()** to create new instances of the command or socket. AWK does not automatically close pipes, sockets, or co-processes when they return EOF.

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, **%F** A floating point number of the form **[−]ddd.dddddd**. If the system library supports it, **%F** is available as well. This is like **%f**, but uses capital letters for special "not a number" and "infinity" values. If **%F** is not available, *gawk* uses **%f**.

%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 (also an integer).

%u An unsigned decimal number (again, an integer).

%s A character string.

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

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

NOTE: When using the integer format-control letters for values that are outside the range of a C **long** integer, *gawk* switches to the **%0f** format specifier. If **−−lint** is provided on the command line *gawk* warns about this. Other versions of *awk* may print invalid values or do something else entirely.

Optional, additional parameters may lie between the **%** and the control letter:

*count***$** Use the *count*'th argument at this point in the formatting. This is called a *positional specifier* and is intended primarily for use in translated versions of format strings, not in the original text of an AWK program. It is a *gawk* extension.

− 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**, **%f** and **%F**, the result always contains 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**, **%f** 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 **%s**, 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 causes their values to be taken from the argument list to **printf** or **sprintf()**. To use a positional specifier with a dynamic width or precision, supply the *count***$** after the * in the format string. For example, **"%3$*2$.*1$s"**.

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). These file names may also be used on the command line to name data files. The filenames are:

/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"
```

The following special filenames may be used with the **|&** co-process operator for creating TCP/IP network connections.

/inet/tcp/*lport***/***rhost***/***rport*	File for TCP/IP connection on local port *lport* to remote host *rhost* on remote port *rport*. Use a port of **0** to have the system pick a port.
/inet/udp/*lport***/***rhost***/***rport*	Similar, but use UDP/IP instead of TCP/IP.
/inet/raw/*lport***/***rhost***/***rport*	Reserved for future use.

Other special filenames provide access to information about the running *gawk* process. **These filenames are now obsolete.** Use the **PROCINFO** array to obtain the information they provide. 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.

Numeric Functions

AWK has the following built-in arithmetic functions:

atan2(*y***,** *x***)**	Returns the arctangent of *y/x* in radians.
cos(*expr***)**	Returns the cosine of *expr*, which is in radians.
exp(*expr***)**	The exponential function.
int(*expr***)**	Truncates to integer.
log(*expr***)**	The natural logarithm function.
rand()	Returns a random number N, between 0 and 1, such that $0 \le N < 1$.
sin(*expr***)**	Returns the sine of *expr*, which is 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 is used. The return value is the previous seed for the random number

generator.

String Functions

Gawk has the following built-in string functions:

asort(*s* [**,** *d*]**)** Returns the number of elements in the source array *s*. The contents of *s* are sorted using *gawk*'s normal rules for comparing values, and the indices of the sorted values of *s* are replaced with sequential integers starting with 1. If the optional destination array *d* is specified, then *s* is first duplicated into *d*, and then *d* is sorted, leaving the indices of the source array *s* unchanged.

asorti(*s* [**,** *d*]**)** Returns the number of elements in the source array *s*. The behavior is the same as that of **asort()**, except that the array *indices* are used for sorting, not the array values. When done, the array is indexed numerically, and the values are those of the original indices. The original values are lost; thus provide a second array if you wish to preserve the original.

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 *t* is not 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 **&**. (This must be typed as **"\\\\&"**; see *GAWK: Effective AWK 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. (This implies that character indices start at one.)

length([*s*]**)** Returns the length of the string *s*, or the length of **$0** if *s* is not supplied. Starting with version 3.1.5, as a non-standard extension, with an array argument, **length()** returns the number of elements in the array.

match(*s***,** *r* [**,** *a*]**)** 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**. Note that the argument order is the same as for the **~** operator: *str* **~** *re*. If array *a* is provided, *a* is cleared and then elements 1 through *n* are filled with the portions of *s* that match the corresponding parenthesized subexpression in *r*. The 0'th element of *a* contains the portion of *s* matched by the entire regular expression *r*. Subscripts **a[***n***, "start"]**, and **a[***n***, "length"]** provide the starting index in the string and length respectively, of each matching substring.

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.

strtonum(*str***)** Examines *str*, and returns its numeric value. If *str* begins with a leading **0**, **strtonum()** assumes that *str* is an octal number. If *str* begins with a leading **0x** or **0X**, **strtonum()** assumes that *str* is a hexadecimal number.

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.

As of version 3.1.5, *gawk* is multibyte aware. This means that **index()**, **length()**, **substr()** and **match()** all work in terms of characters, not bytes.

Time Functions

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

mktime(*datespec***)**

Turns *datespec* into a time stamp of the same form as returned by **systime()**. The *datespec* is a string of the form *YYYY MM DD HH MM SS[DST]*. The contents of the string are six or seven numbers representing respectively the full year including century, the month from 1 to 12, the day of the month from 1 to 31, the hour of the day from 0 to 23, the minute from 0 to 59, and the second from 0 to 60, and an optional daylight saving flag. The values of these numbers need not be within the ranges specified; for example, an hour of −1 means 1 hour before midnight. The origin-zero Gregorian calendar is assumed, with year 0 preceding year 1 and year −1 preceding year 0. The time is assumed to be in the local timezone. If the daylight saving flag is positive, the time is assumed to be daylight saving time; if zero, the time is assumed to be standard time; and if negative (the default), **mktime()** attempts to determine whether daylight saving time is in effect for the specified time. If *datespec* does not contain enough elements or if the resulting time is out of range, **mktime()** returns −1.

strftime([*format* [**,** *timestamp*[**,** *utc-flag*]]]**)**

Formats *timestamp* according to the specification in *format.* If *utc-flag* is present and is non-zero or non-null, the result is in UTC, otherwise the result is in local time. 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) is used. See the specification for the **strftime()** function in ANSI C for the format conversions that are guaranteed to be available.

systime() Returns the current time of day as the number of seconds since the Epoch (1970-01-01 00:00:00 UTC on POSIX systems).

Bit Manipulations Functions

Starting with version 3.1 of *gawk*, the following bit manipulation functions are available. They work by converting double-precision floating point values to **uintmax_t** integers, doing the operation, and then converting the result back to floating point. The functions are:

and(*v1***,** *v2***)**	Return the bitwise AND of the values provided by *v1* and *v2*.
compl(*val***)**	Return the bitwise complement of *val*.
lshift(*val***,** *count***)**	Return the value of *val*, shifted left by *count* bits.
or(*v1***,** *v2***)**	Return the bitwise OR of the values provided by *v1* and *v2*.
rshift(*val***,** *count***)**	Return the value of *val*, shifted right by *count* bits.
xor(*v1***,** *v2***)**	Return the bitwise XOR of the values provided by *v1* and *v2*.

Internationalization Functions

Starting with version 3.1 of *gawk*, the following functions may be used from within your AWK program for translating strings at run-time. For full details, see *GAWK: Effective AWK Programming*.

bindtextdomain(*directory* [**,** *domain*]**)**

Specifies the directory where *gawk* looks for the **.mo** files, in case they will not or cannot be placed in the ‘‘standard’’ locations (e.g., during testing). It returns the directory where *domain* is ‘‘bound.’’

The default *domain* is the value of **TEXTDOMAIN**. If *directory* is the null string (**""**), then **bindtextdomain()** returns the current binding for the given *domain*.

dcgettext(*string* [**,** *domain* [**,** *category*]]**)**
Returns the translation of *string* in text domain *domain* for locale category *category*. The default value for *domain* is the current value of **TEXTDOMAIN**. The default value for *category* is **"LC_MESSAGES"**.

If you supply a value for *category*, it must be a string equal to one of the known locale categories described in *GAWK: Effective AWK Programming*. You must also supply a text domain. Use **TEXTDOMAIN** if you want to use the current domain.

dcngettext(*string1* , *string2* , *number* [**,** *domain* [**,** *category*]]**)**
Returns the plural form used for *number* of the translation of *string1* and *string2* in text domain *domain* for locale category *category*. The default value for *domain* is the current value of **TEXTDOMAIN**. The default value for *category* is **"LC_MESSAGES"**.

If you supply a value for *category*, it must be a string equal to one of the known locale categories described in *GAWK: Effective AWK Programming*. You must also supply a text domain. Use **TEXTDOMAIN** if you want to use the current domain.

USER-DEFINED 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 and 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 avoids 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.

Use **return** *expr* to return a value from a function. The return value is undefined if no value is provided, or if the function returns by “falling off” the end.

If **−−lint** has been provided, *gawk* warns 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**.

DYNAMICALLY LOADING NEW FUNCTIONS

Beginning with version 3.1 of *gawk*, you can dynamically add new built-in functions to the running *gawk* interpreter. The full details are beyond the scope of this manual page; see *GAWK: Effective AWK Programming* for the details.

extension(*object***,** *function***)**
Dynamically link the shared object file named by *object*, and invoke *function* in that object, to perform initialization. These should both be provided as strings. Returns the value returned by *function*.

This function is provided and documented in *GAWK: Effective AWK Programming***, but everything about this feature is likely to change eventually. We STRONGLY recommend that you do not use this feature for anything that you aren’t willing to redo.**

SIGNALS

pgawk accepts two signals. **SIGUSR1** causes it to dump a profile and function call stack to the profile file, which is either **awkprof.out**, or whatever file was named with the **−−profile** option. It then continues to run. **SIGHUP** causes *pgawk* to dump the profile and function call stack and then exit.

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 }
```

Run an external command for particular lines of data:

```
tail -f access_log |
awk '/myhome.html/ { system("nmap " $1 ">> logdir/myhome.html") }'
```

INTERNATIONALIZATION

String constants are sequences of characters enclosed in double quotes. In non-English speaking environments, it is possible to mark strings in the AWK program as requiring translation to the native natural language. Such strings are marked in the AWK program with a leading underscore ("_"). For example,

```
gawk 'BEGIN { print "hello, world" }'
```

always prints **hello, world**. But,

```
gawk 'BEGIN { print _"hello, world" }'
```

might print **bonjour, monde** in France.

There are several steps involved in producing and running a localizable AWK program.

1. Add a **BEGIN** action to assign a value to the **TEXTDOMAIN** variable to set the text domain to a name associated with your program.

   ```
   BEGIN { TEXTDOMAIN = "myprog" }
   ```

This allows *gawk* to find the **.mo** file associated with your program. Without this step, *gawk* uses the **messages** text domain, which likely does not contain translations for your program.

2. Mark all strings that should be translated with leading underscores.
3. If necessary, use the **dcgettext()** and/or **bindtextdomain()** functions in your program, as appropriate.
4. Run **gawk −−gen−po −f myprog.awk > myprog.po** to generate a **.po** file for your program.
5. Provide appropriate translations, and build and install the corresponding **.mo** files.

The internationalization features are described in full detail in *GAWK: Effective AWK Programming*.

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 Laboratories version of *awk*, and are in the POSIX standard.

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, the **−v** option for assigning variables before program execution was added to accommodate applications that depended upon the old behavior. (This feature was agreed upon by both the Bell Laboratories and the 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 warns about but otherwise ignores 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 the Bell Laboratories version); the **tolower()** and **toupper()** built-in functions (from the Bell Laboratories version); and the ANSI C conversion specifications in **printf** (done first in the Bell Laboratories version).

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* issues 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* supports this usage if **−−traditional** has been specified.

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** or **−−posix** options.

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

- No path search is performed for files named via the **−f** option. Therefore the **AWKPATH** environment variable is not special.
- The **\x** escape sequence. (Disabled with **−−posix**.)
- The **fflush()** function. (Disabled with **−−posix**.)
- The ability to continue lines after **?** and **:**. (Disabled with **−−posix**.)
- Octal and hexadecimal constants in AWK programs.
- The **ARGIND**, **BINMODE**, **ERRNO**, **LINT**, **RT** and **TEXTDOMAIN** variables are not special.
- The **IGNORECASE** variable and its side-effects are not available.
- The **FIELDWIDTHS** variable and fixed-width field splitting.
- The **PROCINFO** array is not available.

- The use of **RS** as a regular expression.
- The special file names available for I/O redirection are not recognized.
- The **|&** operator for creating co-processes.
- The ability to split out individual characters using the null string as the value of **FS**, and as the third argument to **split()**.
- The optional second argument to the **close()** function.
- The optional third argument to the **match()** function.
- The ability to use positional specifiers with **printf** and **sprintf()**.
- The ability to pass an array to **length()**.
- The use of **delete** *array* to delete the entire contents of an array.
- The use of **nextfile** to abandon processing of the current input file.
- The **and()**, **asort()**, **asorti()**, **bindtextdomain()**, **compl()**, **dcgettext()**, **dcngettext()**, **gensub()**, **lshift()**, **mktime()**, **or()**, **rshift()**, **strftime()**, **strtonum()**, **systime()** and **xor()** functions.
- Localizable strings.
- Adding new built-in functions dynamically with the **extension()** function.

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 an output file or pipe, respectively. It returns the process's exit status when closing an input pipe. The return value is −1 if the named file, pipe or co-process was not opened with a redirection.

When *gawk* is invoked with the **−−traditional** option, if the *fs* argument to the **−F** option is "t", then **FS** is set to the tab character. Note that typing **gawk −F\t ...** simply causes the shell to quote the "t," and does not pass "\t" to the **−F** option. 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. To really get a tab character as the field separator, it is best to use single quotes: **gawk −F'\t' ...**.

If *gawk* is *configured* with the **−−enable−switch** option to the *configure* command, then it accepts an additional control-flow statement:

> **switch** (*expression*) **{**
> **case** *value*|*regex* **:** *statement*
> ...
> [**default:** *statement*]
> **}**

If *gawk* is configured with the **−−disable−directories-fatal** option, then it will silently skip directories named on the command line. Otherwise, it will do so only if invoked with the **−−traditional** option.

ENVIRONMENT VARIABLES

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

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* issues a warning message to this effect.

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.

GAWK: Effective AWK Programming, Edition 3.0, published by the Free Software Foundation, 2001. The current version of this document is available online at **http://www.gnu.org/software/gawk/manual**.

BUGS

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

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.

AUTHORS

The original version of UNIX *awk* was designed and implemented by Alfred Aho, Peter Weinberger, and Brian Kernighan of Bell Laboratories. 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. Juan M. Guerrero now maintains the OS/2 port. Fred Fish supplied support for the Amiga, and Martin Brown provided the BeOS port. Stephen Davies provided the original Tandem port, and Matthew Woehlke provided changes for Tandem's POSIX-compliant systems.

VERSION INFORMATION

This man page documents *gawk*, version 3.1.6.

BUG REPORTS

If you find a bug in *gawk*, please send electronic mail to **bug-gawk@gnu.org**. Please include your operating system and its revision, the version of *gawk* (from **gawk −−version**), 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 the following 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 see if setting the environment variable **LC_ALL** to **LC_ALL=C** causes things to behave as you expect. If so, it's a locale issue, and may or may not really be a bug. Finally, 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.

If you're using a GNU/Linux system or BSD-based system, you may wish to submit a bug report to the vendor of your distribution. That's fine, but please send a copy to the official email address as well, since there's no guarantee that the bug will be forwarded to the *gawk* maintainer.

ACKNOWLEDGEMENTS

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

COPYING PERMISSIONS

IGAWK

NAME

igawk – gawk with include files

SYNOPSIS

igawk [all *gawk* options] **–f** *program-file* [**––**] file …
igawk [all *gawk* options] [**––**] *program-text* file …

DESCRIPTION

Igawk is a simple shell script that adds the ability to have ‘‘include files’’ to *gawk*(1).

AWK programs for *igawk* are the same as for *gawk*, except that, in addition, you may have lines like

```
@include getopt.awk
```

in your program to include the file **getopt.awk** from either the current directory or one of the other directories in the search path.

OPTIONS

See *gawk*(1) for a full description of the AWK language and the options that *gawk* supports.

EXAMPLES

```
cat << EOF > test.awk
@include getopt.awk

BEGIN {
        while (getopt(ARGC, ARGV, "am:q") != –1)
                ...
}
EOF

igawk –f test.awk
```

SEE ALSO

gawk(1)

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

AUTHOR

Arnold Robbins (**arnold@skeeve.com**).

GAWK

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 ...

pgawk [POSIX or GNU style options] **–f** *program-file* [**––**] file ...
pgawk [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 1003.1 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 Laboratories *awk* extensions, and a number of GNU-specific extensions.

Pgawk is the profiling version of *gawk*. It is identical in every way to *gawk*, except that programs run more slowly, and it automatically produces an execution profile in the file **awkprof.out** when done. See the **––profile** option, below.

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 traditional POSIX one letter options, or 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 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, listed by frequency.

–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 an earlier version of the Bell Laboratories research version of UNIX *awk*. They are ignored by *gawk*, since *gawk* has no pre-defined limits.

–W compat
–W traditional

−−compat
−−traditional
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 standard output and exit successfully.

−W dump-variables[=*file*]
−−dump-variables[=*file*]
Print a sorted list of global variables, their types and final values to *file*. If no *file* is provided, *gawk* uses a file named **awkvars.out** in the current directory.

Having a list of all the global variables is a good way to look for typographical errors in your programs. You would also use this option if you have a large program with a lot of functions, and you want to be sure that your functions don't inadvertently use global variables that you meant to be local. (This is a particularly easy mistake to make with simple variable names like **i**, **j**, and so on.)

−W exec *file*
−−exec *file*
Similar to **−f**, however, this is option is the last one processed. This should be used with **#!** scripts, particularly for CGI applications, to avoid passing in options or source code (!) on the command line from a URL. This option disables command-line variable assignments.

−W gen−po
−−gen−po
Scan and parse the AWK program, and generate a GNU **.po** format file on standard output with entries for all localizable strings in the program. The program itself is not executed. See the GNU *gettext* distribution for more information on **.po** files.

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

−W lint[=*value*]
−−lint[=*value*]
Provide warnings about constructs that are dubious or non-portable to other AWK implementations. With an optional argument of **fatal**, lint warnings become fatal errors. This may be drastic, but its use will certainly encourage the development of cleaner AWK programs. With an optional argument of **invalid**, only warnings about things that are actually invalid are issued. (This is not fully implemented yet.)

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

−W non−decimal−data
−−non−decimal−data
Recognize octal and hexadecimal values in input data. *Use this option with great caution!*

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

- **\x** escape sequences are not recognized.

- Only space and tab act as field separators when **FS** is set to a single space, newline does not.
- You cannot continue lines after **?** and **:**.
- 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 profile[=*prof_file*]
−−profile[=*prof_file*]
Send profiling data to *prof_file*. The default is **awkprof.out**. When run with *gawk*, the profile is just a "pretty printed" version of the program. When run with *pgawk*, the profile contains execution counts of each statement in the program in the left margin and function call counts for each user-defined function.

−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.

−W use−lc−numeric
−−use−lc−numeric
This forces *gawk* to use the locale's decimal point character when parsing input data. Although the POSIX standard requires this behavior, and *gawk* does so when **−−posix** is in effect, the default is to follow traditional behavior and use a period as the decimal point, even in locales where the period is not the decimal point character. This option overrides the default behavior, without the full draconian strictness of the **−−posix** option.

−W version
−−version
Print version information for this particular copy of *gawk* on the standard 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 provides consistency with the argument parsing convention used by most other POSIX programs.

In compatibility mode, any other options are flagged as invalid, 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* reads 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 are 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 separates 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 and/or newlines. (But see the section **POSIX COMPATIBILITY**, below). **NOTE**: The value of **IGNORECASE** (see below) also affects 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* splits 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. 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**) increases the value of **NF**, creates any intervening fields with the null string as their value, and causes 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. Decrementing **NF** causes the values of fields past the new value to be lost, and the value of **$0** to be recomputed, with the fields being separated by the value of **OFS**.

Assigning a value to an existing field causes the whole record to be rebuilt when **$0** is referenced. Similarly, assigning a value to **$0** causes the record to be resplit, creating new values for the fields.

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.
BINMODE	On non-POSIX systems, specifies use of "binary" mode for all file I/O. Numeric values of 1, 2, or 3, specify that input files, output files, or all files, respectively, should use binary I/O. String values of **"r"**, or **"w"** specify that input files, or output files, respectively, should use binary I/O. String values of **"rw"** or **"wr"** specify that all files should use binary I/O. Any other string value is treated as **"rw"**, but generates a warning message.
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.
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. The value is subject to translation in non-English locales.
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.
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 (unless set by **getline**).
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**() built-in functions all ignore case when doing regular expression operations. **NOTE**: Array subscripting is *not* affected. However, the **asort()** and **asorti()** functions are affected. 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. As of *gawk* 3.1.4, the case equivalencies are fully locale-aware, based on the C **<ctype.h>** facilities such as **isalpha**(), and **toupper**().
LINT	Provides dynamic control of the **−−lint** option from within an AWK program. When true, *gawk* prints lint warnings. When false, it does not. When assigned the string value **"fatal"**, lint warnings become fatal errors, exactly like **−−lint=fatal**. Any other true value just prints warnings.
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.
PROCINFO	The elements of this array provide access to information about the running AWK program. On some systems, there may be elements in the array, **"group1"** through **"group***n***"** for some *n*, which is the number of supplementary groups that the process has. Use the **in** operator to test for these elements. The following elements are guaranteed to be available: **PROCINFO["egid"]** the value of the *getegid*(2) system call. **PROCINFO["euid"]** the value of the *geteuid*(2) system call. **PROCINFO["FS"]** **"FS"** if field splitting with **FS** is in effect, or **"FIELDWIDTHS"** if field splitting with **FIELDWIDTHS** is in effect.

PROCINFO["gid"]	the value of the *getgid*(2) system call.
PROCINFO["pgrpid"]	the process group ID of the current process.
PROCINFO["pid"]	the process ID of the current process.
PROCINFO["ppid"]	the parent process ID of the current process.
PROCINFO["uid"]	the value of the *getuid*(2) system call.
PROCINFO["version"]	The version of *gawk*. This is available from version 3.1.4 and later.

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. (This implies that character indices start at one.)
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"**.
TEXTDOMAIN	The text domain of the AWK program; used to find the localized translations for the program's strings.

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 to test 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 *strtod*(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"**.

When operating in POSIX mode (such as with the **−−posix** command line option), beware that locale settings may interfere with the way decimal numbers are treated: the decimal separator of the numbers you are feeding to *gawk* must conform to what your locale would expect, be it a comma (,) or a period (.).

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.

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).

Octal and Hexadecimal Constants

Starting with version 3.1 of *gawk* , you may use C-style octal and hexadecimal constants in your AWK program source code. For example, the octal value **011** is equal to decimal **9**, and the hexadecimal value **0x11** is equal to decimal 17.

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 regular expression constants. Thus, **/a\52b/** is equivalent to **/a\*b/**.

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 is 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...*.
*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 regular expression *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 valid in regular expressions.

Character classes are a 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 regular expression *inside* the brackets of a character list. Character classes consist of **[:**, a keyword denoting the class, and **:]**. The character classes defined by the POSIX standard are:

[: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, and if your character set collated differently from ASCII, this might not even match the ASCII alphanumeric characters. With the POSIX character classes, you can write **/[[:alnum:]]/**, and this matches the alphabetic and numeric characters in your character set, no matter what it is.

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 “è” are equivalent.)

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

Equivalence Classes
: An equivalence class is a locale-specific name for a list of characters that are equivalent. The name is enclosed in **[=** and **=]**. For example, the name **e** might be used to represent all of “e,” “é,” and “è.” In this case, **[[=e=]]** is a regular expression that matches any of **e**, **é**, or **è**.

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 regular expression libraries.

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

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

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

––traditional
: Traditional Unix *awk* regular expressions 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 regular expression metacharacters.

−−re−interval
Allow interval expressions in regular expressions, 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.
| |&	Piped I/O for **getline**, **print**, and **printf**.
< > **<= >=** **!= ==**	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* [**,** *how*]**)**	Close file, pipe or co-process. The optional *how* should only be used when closing one end of a two-way pipe to a co-process. It must be a string value, either **"to"** or **"from"**.
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 **NR**, **FNR**.
getline *var* **<***file*	Set *var* from next record of *file*.
command **| getline** [*var*]	Run *command* piping the output either into **$0** or *var*, as above.
command **|& getline** [*var*]	Run *command* as a co-process piping the output either into **$0** or *var*, as above. Co-processes are a *gawk* extension. (*command* can also be a socket. See the subsection **Special File Names**, below.)
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.
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.

Additional output redirections are allowed for **print** and **printf**.

print ... >> *file*
Appends output to the *file*.

print ... | *command*
Writes on a pipe.

print ... |& *command*
Sends data to a co-process or socket. (See also the subsection **Special File Names**, below.)

The **getline** command returns 0 on end of file and −1 on an error. Upon an error, **ERRNO** contains a string describing the problem.

NOTE: If using a pipe, co-process, or socket to **getline**, or from **print** or **printf** within a loop, you *must* use **close()** to create new instances of the command or socket. AWK does not automatically close pipes, sockets, or co-processes when they return EOF.

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, **%F**	A floating point number of the form **[−]ddd.dddddd**. If the system library supports it, **%F** is available as well. This is like **%f**, but uses capital letters for special "not a number" and "infinity" values. If **%F** is not available, *gawk* uses **%f**.
%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 (also an integer).
%u	An unsigned decimal number (again, an integer).
%s	A character string.
%x, **%X**	An unsigned hexadecimal number (an integer). The **%X** format uses **ABCDEF** instead of **abcdef**.
%%	A single **%** character; no argument is converted.

NOTE: When using the integer format-control letters for values that are outside the range of a C **long** integer, *gawk* switches to the **%0f** format specifier. If **−−lint** is provided on the command line *gawk* warns about this. Other versions of *awk* may print invalid values or do something else entirely.

Optional, additional parameters may lie between the **%** and the control letter:

count$	Use the *count*'th argument at this point in the formatting. This is called a *positional specifier* and is intended primarily for use in translated versions of format strings, not in the original text of an AWK program. It is a *gawk* extension.
−	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**, **%f** and **%F**, the result always contains 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**, **%f** 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 **%s**, 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 causes their values to be taken from the argument list to **printf** or **sprintf()**. To use a positional specifier with a dynamic width or precision, supply the *count*$ after the * in the format string. For example, **"%3$*2$.*1$s"**.

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). These file names may also be used on the command line to name data files. The filenames are:

/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"
```

The following special filenames may be used with the **|&** co-process operator for creating TCP/IP network connections.

/inet/tcp/*lport***/***rhost***/***rport*	File for TCP/IP connection on local port *lport* to remote host *rhost* on remote port *rport*. Use a port of **0** to have the system pick a port.
/inet/udp/*lport***/***rhost***/***rport*	Similar, but use UDP/IP instead of TCP/IP.
/inet/raw/*lport***/***rhost***/***rport*	Reserved for future use.

Other special filenames provide access to information about the running *gawk* process. **These filenames are now obsolete.** Use the **PROCINFO** array to obtain the information they provide. 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.

Numeric Functions

AWK has the following built-in arithmetic functions:

atan2(*y***,** *x***)**	Returns the arctangent of *y/x* in radians.
cos(*expr***)**	Returns the cosine of *expr*, which is in radians.
exp(*expr***)**	The exponential function.
int(*expr***)**	Truncates to integer.
log(*expr***)**	The natural logarithm function.
rand()	Returns a random number N, between 0 and 1, such that $0 \le N < 1$.
sin(*expr***)**	Returns the sine of *expr*, which is 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 is used. The return value is the previous seed for the random number

generator.

String Functions

Gawk has the following built-in string functions:

asort(*s* [**,** *d*]) Returns the number of elements in the source array *s*. The contents of *s* are sorted using *gawk*'s normal rules for comparing values, and the indices of the sorted values of *s* are replaced with sequential integers starting with 1. If the optional destination array *d* is specified, then *s* is first duplicated into *d*, and then *d* is sorted, leaving the indices of the source array *s* unchanged.

asorti(*s* [**,** *d*]) Returns the number of elements in the source array *s*. The behavior is the same as that of **asort()**, except that the array *indices* are used for sorting, not the array values. When done, the array is indexed numerically, and the values are those of the original indices. The original values are lost; thus provide a second array if you wish to preserve the original.

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 *t* is not 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 **&**. (This must be typed as **"\\\\&"**; see *GAWK: Effective AWK 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. (This implies that character indices start at one.)

length([*s*]) Returns the length of the string *s*, or the length of **$0** if *s* is not supplied. Starting with version 3.1.5, as a non-standard extension, with an array argument, **length()** returns the number of elements in the array.

match(*s***,** *r* [**,** *a*]) 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**. Note that the argument order is the same as for the **~** operator: *str* **~** *re*. If array *a* is provided, *a* is cleared and then elements 1 through *n* are filled with the portions of *s* that match the corresponding parenthesized subexpression in *r*. The 0'th element of *a* contains the portion of *s* matched by the entire regular expression *r*. Subscripts **a[***n***, "start"]**, and **a[***n***, "length"]** provide the starting index in the string and length respectively, of each matching substring.

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.

strtonum(*str***)** Examines *str*, and returns its numeric value. If *str* begins with a leading **0**, **strtonum()** assumes that *str* is an octal number. If *str* begins with a leading **0x** or **0X**, **strtonum()** assumes that *str* is a hexadecimal number.

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.

As of version 3.1.5, *gawk* is multibyte aware. This means that **index()**, **length()**, **substr()** and **match()** all work in terms of characters, not bytes.

Time Functions

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

mktime(*datespec***)**
Turns *datespec* into a time stamp of the same form as returned by **systime()**. The *datespec* is a string of the form *YYYY MM DD HH MM SS[DST]*. The contents of the string are six or seven numbers representing respectively the full year including century, the month from 1 to 12, the day of the month from 1 to 31, the hour of the day from 0 to 23, the minute from 0 to 59, and the second from 0 to 60, and an optional daylight saving flag. The values of these numbers need not be within the ranges specified; for example, an hour of −1 means 1 hour before midnight. The origin-zero Gregorian calendar is assumed, with year 0 preceding year 1 and year −1 preceding year 0. The time is assumed to be in the local timezone. If the daylight saving flag is positive, the time is assumed to be daylight saving time; if zero, the time is assumed to be standard time; and if negative (the default), **mktime()** attempts to determine whether daylight saving time is in effect for the specified time. If *datespec* does not contain enough elements or if the resulting time is out of range, **mktime()** returns −1.

strftime([*format* [**,** *timestamp*[**,** *utc-flag*]]]**)**
Formats *timestamp* according to the specification in *format.* If *utc-flag* is present and is non-zero or non-null, the result is in UTC, otherwise the result is in local time. 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) is used. See the specification for the **strftime()** function in ANSI C for the format conversions that are guaranteed to be available.

systime() Returns the current time of day as the number of seconds since the Epoch (1970-01-01 00:00:00 UTC on POSIX systems).

Bit Manipulations Functions

Starting with version 3.1 of *gawk*, the following bit manipulation functions are available. They work by converting double-precision floating point values to **uintmax_t** integers, doing the operation, and then converting the result back to floating point. The functions are:

and(*v1***,** *v2***)**	Return the bitwise AND of the values provided by *v1* and *v2*.
compl(*val***)**	Return the bitwise complement of *val*.
lshift(*val***,** *count***)**	Return the value of *val*, shifted left by *count* bits.
or(*v1***,** *v2***)**	Return the bitwise OR of the values provided by *v1* and *v2*.
rshift(*val***,** *count***)**	Return the value of *val*, shifted right by *count* bits.
xor(*v1***,** *v2***)**	Return the bitwise XOR of the values provided by *v1* and *v2*.

Internationalization Functions

Starting with version 3.1 of *gawk*, the following functions may be used from within your AWK program for translating strings at run-time. For full details, see *GAWK: Effective AWK Programming*.

bindtextdomain(*directory* [**,** *domain*]**)**
Specifies the directory where *gawk* looks for the **.mo** files, in case they will not or cannot be placed in the "standard" locations (e.g., during testing). It returns the directory where *domain* is "bound."

The default *domain* is the value of **TEXTDOMAIN**. If *directory* is the null string (**""**), then **bindtextdomain()** returns the current binding for the given *domain*.

dcgettext(*string* [**,** *domain* [**,** *category*]]**)**
Returns the translation of *string* in text domain *domain* for locale category *category*. The default value for *domain* is the current value of **TEXTDOMAIN**. The default value for *category* is **"LC_MESSAGES"**.

If you supply a value for *category*, it must be a string equal to one of the known locale categories described in *GAWK: Effective AWK Programming*. You must also supply a text domain. Use **TEXTDOMAIN** if you want to use the current domain.

dcngettext(*string1* , *string2* , *number* [**,** *domain* [**,** *category*]]**)**
Returns the plural form used for *number* of the translation of *string1* and *string2* in text domain *domain* for locale category *category*. The default value for *domain* is the current value of **TEXTDOMAIN**. The default value for *category* is **"LC_MESSAGES"**.

If you supply a value for *category*, it must be a string equal to one of the known locale categories described in *GAWK: Effective AWK Programming*. You must also supply a text domain. Use **TEXTDOMAIN** if you want to use the current domain.

USER-DEFINED 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 and 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 avoids 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.

Use **return** *expr* to return a value from a function. The return value is undefined if no value is provided, or if the function returns by "falling off" the end.

If **−−lint** has been provided, *gawk* warns 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**.

DYNAMICALLY LOADING NEW FUNCTIONS

Beginning with version 3.1 of *gawk*, you can dynamically add new built-in functions to the running *gawk* interpreter. The full details are beyond the scope of this manual page; see *GAWK: Effective AWK Programming* for the details.

extension(*object***,** *function***)**
Dynamically link the shared object file named by *object*, and invoke *function* in that object, to perform initialization. These should both be provided as strings. Returns the value returned by *function*.

This function is provided and documented in *GAWK: Effective AWK Programming***, but everything about this feature is likely to change eventually. We STRONGLY recommend that you do not use this feature for anything that you aren't willing to redo.**

SIGNALS

pgawk accepts two signals. **SIGUSR1** causes it to dump a profile and function call stack to the profile file, which is either **awkprof.out**, or whatever file was named with the **−−profile** option. It then continues to run. **SIGHUP** causes *pgawk* to dump the profile and function call stack and then exit.

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 }
```

Run an external command for particular lines of data:

```
tail -f access_log |
awk '/myhome.html/ { system("nmap " $1 ">> logdir/myhome.html") }'
```

INTERNATIONALIZATION

String constants are sequences of characters enclosed in double quotes. In non-English speaking environments, it is possible to mark strings in the AWK program as requiring translation to the native natural language. Such strings are marked in the AWK program with a leading underscore (“_”). For example,

```
gawk 'BEGIN { print "hello, world" }'
```

always prints **hello, world**. But,

```
gawk 'BEGIN { print _"hello, world" }'
```

might print **bonjour, monde** in France.

There are several steps involved in producing and running a localizable AWK program.

1. Add a **BEGIN** action to assign a value to the **TEXTDOMAIN** variable to set the text domain to a name associated with your program.

```
BEGIN { TEXTDOMAIN = "myprog" }
```

This allows *gawk* to find the **.mo** file associated with your program. Without this step, *gawk* uses the **messages** text domain, which likely does not contain translations for your program.

2. Mark all strings that should be translated with leading underscores.
3. If necessary, use the **dcgettext()** and/or **bindtextdomain()** functions in your program, as appropriate.
4. Run **gawk −−gen−po −f myprog.awk > myprog.po** to generate a **.po** file for your program.
5. Provide appropriate translations, and build and install the corresponding **.mo** files.

The internationalization features are described in full detail in *GAWK: Effective AWK Programming*.

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 Laboratories version of *awk*, and are in the POSIX standard.

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, the **−v** option for assigning variables before program execution was added to accommodate applications that depended upon the old behavior. (This feature was agreed upon by both the Bell Laboratories and the 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 warns about but otherwise ignores 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 the Bell Laboratories version); the **tolower()** and **toupper()** built-in functions (from the Bell Laboratories version); and the ANSI C conversion specifications in **printf** (done first in the Bell Laboratories version).

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* issues 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* supports this usage if **−−traditional** has been specified.

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** or **−−posix** options.

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

- No path search is performed for files named via the **−f** option. Therefore the **AWKPATH** environment variable is not special.
- The **\x** escape sequence. (Disabled with **−−posix**.)
- The **fflush()** function. (Disabled with **−−posix**.)
- The ability to continue lines after **?** and **:**. (Disabled with **−−posix**.)
- Octal and hexadecimal constants in AWK programs.
- The **ARGIND**, **BINMODE**, **ERRNO**, **LINT**, **RT** and **TEXTDOMAIN** variables are not special.
- The **IGNORECASE** variable and its side-effects are not available.
- The **FIELDWIDTHS** variable and fixed-width field splitting.
- The **PROCINFO** array is not available.

- The use of **RS** as a regular expression.
- The special file names available for I/O redirection are not recognized.
- The **|&** operator for creating co-processes.
- The ability to split out individual characters using the null string as the value of **FS**, and as the third argument to **split()**.
- The optional second argument to the **close()** function.
- The optional third argument to the **match()** function.
- The ability to use positional specifiers with **printf** and **sprintf()**.
- The ability to pass an array to **length()**.
- The use of **delete** *array* to delete the entire contents of an array.
- The use of **nextfile** to abandon processing of the current input file.
- The **and()**, **asort()**, **asorti()**, **bindtextdomain()**, **compl()**, **dcgettext()**, **dcngettext()**, **gensub()**, **lshift()**, **mktime()**, **or()**, **rshift()**, **strftime()**, **strtonum()**, **systime()** and **xor()** functions.
- Localizable strings.
- Adding new built-in functions dynamically with the **extension()** function.

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 an output file or pipe, respectively. It returns the process's exit status when closing an input pipe. The return value is −1 if the named file, pipe or co-process was not opened with a redirection.

When *gawk* is invoked with the **−−traditional** option, if the *fs* argument to the **−F** option is "t", then **FS** is set to the tab character. Note that typing **gawk −F\t ...** simply causes the shell to quote the "t," and does not pass "\t" to the **−F** option. 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. To really get a tab character as the field separator, it is best to use single quotes: **gawk −F'\t' ...**.

If *gawk* is *configured* with the **−−enable−switch** option to the *configure* command, then it accepts an additional control-flow statement:

```
switch (expression) {
case value|regex : statement
...
[ default: statement ]
}
```

If *gawk* is configured with the **−−disable−directories-fatal** option, then it will silently skip directories named on the command line. Otherwise, it will do so only if invoked with the **−−traditional** option.

ENVIRONMENT VARIABLES

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

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* issues a warning message to this effect.

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.

GAWK: Effective AWK Programming, Edition 3.0, published by the Free Software Foundation, 2001. The current version of this document is available online at **http://www.gnu.org/software/gawk/manual**.

BUGS

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

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.

AUTHORS

The original version of UNIX *awk* was designed and implemented by Alfred Aho, Peter Weinberger, and Brian Kernighan of Bell Laboratories. 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. Juan M. Guerrero now maintains the OS/2 port. Fred Fish supplied support for the Amiga, and Martin Brown provided the BeOS port. Stephen Davies provided the original Tandem port, and Matthew Woehlke provided changes for Tandem's POSIX-compliant systems.

VERSION INFORMATION

This man page documents *gawk*, version 3.1.6.

BUG REPORTS

If you find a bug in *gawk*, please send electronic mail to **bug-gawk@gnu.org**. Please include your operating system and its revision, the version of *gawk* (from **gawk −−version**), 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 the following 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 see if setting the environment variable **LC_ALL** to **LC_ALL=C** causes things to behave as you expect. If so, it's a locale issue, and may or may not really be a bug. Finally, 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.

If you're using a GNU/Linux system or BSD-based system, you may wish to submit a bug report to the vendor of your distribution. That's fine, but please send a copy to the official email address as well, since there's no guarantee that the bug will be forwarded to the *gawk* maintainer.

ACKNOWLEDGEMENTS

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

COPYING PERMISSIONS

AWK

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

awk − pattern scanning and processing language

SYNOPSIS

awk [-F *ERE***][-v** *assignment***] ...** *program* **[***argument* **...]**

awk [-F *ERE***] -f** *progfile* **... [-v** *assignment***] ...[***argument* **...]**

DESCRIPTION

The *awk* utility shall execute programs written in the *awk* programming language, which is specialized for textual data manipulation. An *awk* program is a sequence of patterns and corresponding actions. When input is read that matches a pattern, the action associated with that pattern is carried out.

Input shall be interpreted as a sequence of records. By default, a record is a line, less its terminating <newline>, but this can be changed by using the **RS** built-in variable. Each record of input shall be matched in turn against each pattern in the program. For each pattern matched, the associated action shall be executed.

The *awk* utility shall interpret each input record as a sequence of fields where, by default, a field is a string of non- <blank>s. This default white-space field delimiter can be changed by using the **FS** built-in variable or **-F** *ERE*. The *awk* utility shall denote the first field in a record $1, the second $2, and so on. The symbol $0 shall refer to the entire record; setting any other field causes the re-evaluation of $0. Assigning to $0 shall reset the values of all other fields and the **NF** built-in variable.

OPTIONS

The *awk* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

-F *ERE*

Define the input field separator to be the extended regular expression *ERE*, before any input is read; see Regular Expressions .

-f *progfile*

Specify the pathname of the file *progfile* containing an *awk* program. If multiple instances of this option are specified, the concatenation of the files specified as *progfile* in the order specified shall be the *awk* program. The *awk* program can alternatively be specified in the command line as a single argument.

-v *assignment*

The application shall ensure that the *assignment* argument is in the same form as an *assignment* operand. The specified variable assignment shall occur prior to executing the *awk* program, including the actions associated with **BEGIN** patterns (if any). Multiple occurrences of this option can be specified.

OPERANDS

The following operands shall be supported:

program

If no **-f** option is specified, the first operand to *awk* shall be the text of the *awk* program. The application shall supply the *program* operand as a single argument to *awk*. If the text does not end in a <newline>, *awk* shall interpret the text as if it did.

argument

Either of the following two types of *argument* can be intermixed:

file

A pathname of a file that contains the input to be read, which is matched against the set of patterns in the program. If no *file* operands are specified, or if a *file* operand is **'-'**, the standard input shall be used.

assignment
An operand that begins with an underscore or alphabetic character from the portable character set (see the table in the Base Definitions volume of IEEE Std 1003.1-2001, Section 6.1, Portable Character Set), followed by a sequence of underscores, digits, and alphabetics from the portable character set, followed by the **'='** character, shall specify a variable assignment rather than a pathname. The characters before the **'='** represent the name of an *awk* variable; if that name is an *awk* reserved word (see Grammar) the behavior is undefined. The characters following the equal sign shall be interpreted as if they appeared in the *awk* program preceded and followed by a double-quote (**')'** character, as a **STRING** token (see Grammar), except that if the last character is an unescaped backslash, it shall be interpreted as a literal backslash rather than as the first character of the sequence **"\""** . The variable shall be assigned the value of that **STRING** token and, if appropriate, shall be considered a *numeric string* (see Expressions in awk), the variable shall also be assigned its numeric value. Each such variable assignment shall occur just prior to the processing of the following *file*, if any. Thus, an assignment before the first *file* argument shall be executed after the **BEGIN** actions (if any), while an assignment after the last *file* argument shall occur before the **END** actions (if any). If there are no *file* arguments, assignments shall be executed before processing the standard input.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is **'-'** ; see the INPUT FILES section. If the *awk* program contains no actions and no patterns, but is otherwise a valid *awk* program, standard input and any *file* operands shall not be read and *awk* shall exit with a return status of zero.

INPUT FILES

Input files to the *awk* program from any of the following sources shall be text files:

* Any *file* operands or their equivalents, achieved by modifying the *awk* variables **ARGV** and **ARGC**
* Standard input in the absence of any *file* operands
* Arguments to the **getline** function

Whether the variable **RS** is set to a value other than a <newline> or not, for these files, implementations shall support records terminated with the specified separator up to {LINE_MAX} bytes and may support longer records.

If **-f** *progfile* is specified, the application shall ensure that the files named by each of the *progfile* option-arguments are text files and their concatenation, in the same order as they appear in the arguments, is an *awk* program.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *awk*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL
If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE
Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions and in comparisons of string values.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the

behavior of character classes within regular expressions, the identification of characters as letters, and the mapping of uppercase and lowercase characters for the **toupper** and **tolower** functions.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

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).

NLSPATH
Determine the location of message catalogs for the processing of *LC_MESSAGES* .

PATH Determine the search path when looking for commands executed by *system*(*expr*), or input and output pipes; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables.

In addition, all environment variables shall be visible via the *awk* variable **ENVIRON**.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The nature of the output files depends on the *awk* program.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The nature of the output files depends on the *awk* program.

EXTENDED DESCRIPTION

Overall Program Structure

An *awk* program is composed of pairs of the form:

pattern **{** *action* **}**

Either the pattern or the action (including the enclosing brace characters) can be omitted.

A missing pattern shall match any record of input, and a missing action shall be equivalent to:

{ print }

Execution of the *awk* program shall start by first executing the actions associated with all **BEGIN** patterns in the order they occur in the program. Then each *file* operand (or standard input if no files were specified) shall be processed in turn by reading data from the file until a record separator is seen (<newline> by default). Before the first reference to a field in the record is evaluated, the record shall be split into fields, according to the rules in Regular Expressions, using the value of **FS** that was current at the time the record was read. Each pattern in the program then shall be evaluated in the order of occurrence, and the action associated with each pattern that matches the current record executed. The action for a matching pattern shall be executed before evaluating subsequent patterns. Finally, the actions associated with all **END** patterns shall be executed in the order they occur in the program.

Expressions in awk

Expressions describe computations used in *patterns* and *actions*. In the following table, valid expression operations are given in groups from highest precedence first to lowest precedence last, with equal-precedence operators grouped between horizontal lines. In expression evaluation, where the grammar is formally ambiguous, higher precedence operators shall be evaluated before lower precedence operators. In this table *expr*, *expr1*, *expr2*, and *expr3* represent any expression, while lvalue represents any entity that can be assigned to (that is, on the left side of an assignment operator). The precise syntax of expressions is given in Grammar .

Table: Expressions in Decreasing Precedence in *awk*

Syntax	Name	Type of Result	Associativity
(expr)	Grouping	Type of expr	N/A
$expr	Field reference	String	N/A
++ lvalue	Pre-increment	Numeric	N/A
-- lvalue	Pre-decrement	Numeric	N/A
lvalue ++	Post-increment	Numeric	N/A
lvalue --	Post-decrement	Numeric	N/A
expr ^ expr	Exponentiation	Numeric	Right
! expr	Logical not	Numeric	N/A
+ expr	Unary plus	Numeric	N/A
- expr	Unary minus	Numeric	N/A
expr * expr	Multiplication	Numeric	Left
expr / expr	Division	Numeric	Left
expr % expr	Modulus	Numeric	Left
expr + expr	Addition	Numeric	Left
expr - expr	Subtraction	Numeric	Left
expr expr	String concatenation	String	Left
expr < expr	Less than	Numeric	None
expr <= expr	Less than or equal to	Numeric	None
expr != expr	Not equal to	Numeric	None
expr == expr	Equal to	Numeric	None
expr > expr	Greater than	Numeric	None
expr >= expr	Greater than or equal to	Numeric	None
expr ~ expr	ERE match	Numeric	None
expr !~ expr	ERE non-match	Numeric	None
expr in array	Array membership	Numeric	Left
(index) in array	Multi-dimension array membership	Numeric	Left
expr && expr	Logical AND	Numeric	Left
expr || expr	Logical OR	Numeric	Left
expr1 ? expr2 : expr3	Conditional expression	Type of selected expr2 or expr3	Right
lvalue ^= expr	Exponentiation assignment	Numeric	Right
lvalue %= expr	Modulus assignment	Numeric	Right
lvalue *= expr	Multiplication assignment	Numeric	Right
lvalue /= expr	Division assignment	Numeric	Right
lvalue += expr	Addition assignment	Numeric	Right
lvalue -= expr	Subtraction assignment	Numeric	Right
lvalue = expr	Assignment	Type of expr	Right

Each expression shall have either a string value, a numeric value, or both. Except as stated for specific contexts, the value of an expression shall be implicitly converted to the type needed for the context in which it is used. A string value shall be converted to a numeric value by the equivalent of the following calls to functions defined by the ISO C standard:

```
setlocale(LC_NUMERIC, "");
numeric_value = atof(string_value);
```

A numeric value that is exactly equal to the value of an integer (see *Concepts Derived from the ISO C Standard*) shall be converted to a string by the equivalent of a call to the **sprintf** function (see String Functions) with the string **"%d"** as the *fmt* argument and the numeric value being converted as the first and only *expr* argument. Any other numeric value shall be converted to a string by the equivalent of a call to the **sprintf** function with the value of the variable **CONVFMT** as the *fmt* argument and the numeric value being converted as the first and only *expr* argument. The result of the conversion is unspecified if the value of **CONVFMT** is not a floating-point format specification. This volume of

IEEE Std 1003.1-2001 specifies no explicit conversions between numbers and strings. An application can force an expression to be treated as a number by adding zero to it, or can force it to be treated as a string by concatenating the null string ("") to it.

A string value shall be considered a *numeric string* if it comes from one of the following:

1. Field variables
2. Input from the *getline*() function
3. **FILENAME**
4. **ARGV** array elements
5. **ENVIRON** array elements
6. Array elements created by the *split*() function
7. A command line variable assignment
8. Variable assignment from another numeric string variable

and after all the following conversions have been applied, the resulting string would lexically be recognized as a **NUMBER** token as described by the lexical conventions in Grammar :

* All leading and trailing <blank>s are discarded.
* If the first non- <blank> is **'+'** or **'-'**, it is discarded.
* Changing each occurrence of the decimal point character from the current locale to a period.

If a **'-'** character is ignored in the preceding description, the numeric value of the *numeric string* shall be the negation of the numeric value of the recognized **NUMBER** token. Otherwise, the numeric value of the *numeric string* shall be the numeric value of the recognized **NUMBER** token. Whether or not a string is a *numeric string* shall be relevant only in contexts where that term is used in this section.

When an expression is used in a Boolean context, if it has a numeric value, a value of zero shall be treated as false and any other value shall be treated as true. Otherwise, a string value of the null string shall be treated as false and any other value shall be treated as true. A Boolean context shall be one of the following:

* The first subexpression of a conditional expression
* An expression operated on by logical NOT, logical AND, or logical OR
* The second expression of a **for** statement
* The expression of an **if** statement
* The expression of the **while** clause in either a **while** or **do**... **while** statement
* An expression used as a pattern (as in Overall Program Structure)

All arithmetic shall follow the semantics of floating-point arithmetic as specified by the ISO C standard (see *Concepts Derived from the ISO C Standard*).

The value of the expression:

```
expr1 ^ expr2
```

shall be equivalent to the value returned by the ISO C standard function call:

```
pow(expr1, expr2)
```

The expression:

lvalue ^= *expr*

shall be equivalent to the ISO C standard expression:

lvalue = pow(lvalue, *expr***)**

except that lvalue shall be evaluated only once. The value of the expression:

expr1 % *expr2*

shall be equivalent to the value returned by the ISO C standard function call:

fmod(*expr1***,** *expr2***)**

The expression:

lvalue %= *expr*

shall be equivalent to the ISO C standard expression:

lvalue = fmod(lvalue, *expr***)**

except that lvalue shall be evaluated only once.

Variables and fields shall be set by the assignment statement:

lvalue = *expression*

and the type of *expression* shall determine the resulting variable type. The assignment includes the arithmetic assignments ("+=", "-=", "*=", "/=", "%=", "^=", "++", "--") all of which shall produce a numeric result. The left-hand side of an assignment and the target of increment and decrement operators can be one of a variable, an array with index, or a field selector.

The *awk* language supplies arrays that are used for storing numbers or strings. Arrays need not be declared. They shall initially be empty, and their sizes shall change dynamically. The subscripts, or element identifiers, are strings, providing a type of associative array capability. An array name followed by a subscript within square brackets can be used as an lvalue and thus as an expression, as described in the grammar; see Grammar . Unsubscripted array names can be used in only the following contexts:

* A parameter in a function definition or function call
* The **NAME** token following any use of the keyword **in** as specified in the grammar (see Grammar); if the name used in this context is not an array name, the behavior is undefined

A valid array *index* shall consist of one or more comma-separated expressions, similar to the way in which multi-dimensional arrays are indexed in some programming languages. Because *awk* arrays are really one-dimensional, such a comma-separated list shall be converted to a single string by concatenating the string values of the separate expressions, each separated from the other by the value of the **SUBSEP** variable. Thus, the following two index operations shall be equivalent:

var[*expr1*, *expr2*, **...** *exprn*]

var[*expr1* **SUBSEP** *expr2* **SUBSEP ... SUBSEP** *exprn*]

The application shall ensure that a multi-dimensioned *index* used with the **in** operator is parenthesized. The **in** operator, which tests for the existence of a particular array element, shall not cause that element to exist. Any other reference to a nonexistent array element shall automatically create it.

Comparisons (with the **'<'**, **"<="**, **"!="**, **"=="**, **'>'**, and **">="** operators) shall be made numerically if both operands are numeric, if one is numeric and the other has a string value that is a numeric string, or if one is numeric and the other has the uninitialized value. Otherwise, operands shall be converted to strings as required and a string comparison shall be made using the locale-specific collation sequence. The value of the comparison expression shall be 1 if the relation is true, or 0 if the relation is false.

Variables and Special Variables

Variables can be used in an *awk* program by referencing them. With the exception of function parameters (see User-Defined Functions), they are not explicitly declared. Function parameter names shall be local to the function; all other variable names shall be global. The same name shall not be used as both a function parameter name and as the name of a function or a special *awk* variable. The same name shall not be used both as a variable name with global scope and as the name of a function. The same name shall not be used within the same scope both as a scalar variable and as an array. Uninitialized variables, including scalar variables, array elements, and field variables, shall have an uninitialized value. An uninitialized value shall have both a numeric value of zero and a string value of the empty string. Evaluation of variables with an uninitialized value, to either string or numeric, shall be determined by the context in which they are used.

Field variables shall be designated by a **'$'** followed by a number or numerical expression. The effect of the field number *expression* evaluating to anything other than a non-negative integer is unspecified; uninitialized variables or string values need not be converted to numeric values in this context. New field variables can be created by assigning a value to them. References to nonexistent fields (that is, fields after $**NF**), shall evaluate to the uninitialized value. Such references shall not create new fields. However, assigning to a nonexistent field (for example, $(**NF**+2)=5) shall increase the value of **NF**; create any intervening fields with the uninitialized value; and cause the value of $0 to be recomputed, with the fields being separated by the value of **OFS**. Each field variable shall have a string value or an uninitialized value when created. Field variables shall have the uninitialized value when created from $0 using **FS** and the variable does not contain any characters. If appropriate, the field variable shall be considered a numeric string (see Expressions in awk).

Implementations shall support the following other special variables that are set by *awk*:

ARGC The number of elements in the **ARGV** array.

ARGV An array of command line arguments, excluding options and the *program* argument, numbered from zero to **ARGC**-1.

The arguments in **ARGV** can be modified or added to; **ARGC** can be altered. As each input file ends, *awk* shall treat the next non-null element of **ARGV**, up to the current value of **ARGC**-1, inclusive, as the name of the next input file. Thus, setting an element of **ARGV** to null means that it shall not be treated as an input file. The name **'-'** indicates the standard input. If an argument matches the format of an *assignment* operand, this argument shall be treated as an *assignment* rather than a *file* argument.

CONVFMT
The **printf** format for converting numbers to strings (except for output statements, where **OFMT** is used); **"%.6g"** by default.

ENVIRON
An array representing the value of the environment, as described in the *exec* functions defined in the System Interfaces volume of IEEE Std 1003.1-2001. The indices of the array shall be strings consisting of the names of the environment variables, and the value of each array element shall be a string consisting of the value of that variable. If appropriate, the environment variable shall be considered a *numeric string* (see Expressions in awk); the array element shall also have its numeric value.

In all cases where the behavior of *awk* is affected by environment variables (including the environment of any commands that *awk* executes via the **system** function or via pipeline redirections with the **print** statement, the **printf** statement, or the **getline** function), the environment used shall be the environment at the time *awk* began executing; it is implementation-defined whether any modification of **ENVIRON** affects this environment.

FILENAME
A pathname of the current input file. Inside a **BEGIN** action the value is undefined. Inside an **END** action the value shall be the name of the last input file processed.

FNR The ordinal number of the current record in the current file. Inside a **BEGIN** action the value shall be zero. Inside an **END** action the value shall be the number of the last record processed in the last file processed.

FS Input field separator regular expression; a <space> by default.

NF The number of fields in the current record. Inside a **BEGIN** action, the use of **NF** is undefined unless a **getline** function without a *var* argument is executed previously. Inside an **END** action, **NF** shall retain the value it had for the last record read, unless a subsequent, redirected, **getline** function without a *var* argument is performed prior to entering the **END** action.

NR The ordinal number of the current record from the start of input. Inside a **BEGIN** action the value shall be zero. Inside an **END** action the value shall be the number of the last record processed.

OFMT The **printf** format for converting numbers to strings in output statements (see Output Statements); **"%.6g"** by default. The result of the conversion is unspecified if the value of **OFMT** is not a floating-point format specification.

OFS The **print** statement output field separation; <space> by default.

ORS The **print** statement output record separator; a <newline> by default.

RLENGTH
The length of the string matched by the **match** function.

RS The first character of the string value of **RS** shall be the input record separator; a <newline> by default. If **RS** contains more than one character, the results are unspecified. If **RS** is null, then records are separated by sequences consisting of a <newline> plus one or more blank lines, leading or trailing blank lines shall not result in empty records at the beginning or end of the input, and a <newline> shall always be a field separator, no matter what the value of **FS** is.

RSTART
The starting position of the string matched by the **match** function, numbering from 1. This shall always be equivalent to the return value of the **match** function.

SUBSEP
The subscript separator string for multi-dimensional arrays; the default value is implementation-defined.

Regular Expressions

The *awk* utility shall make use of the extended regular expression notation (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 9.4, Extended Regular Expressions) except that it shall allow the use of C-language conventions for escaping special characters within the EREs, as specified in the table in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 5, File Format Notation (**'\\'**, **'\a'**, **'\b'**, **'\f'**, **'\n'**, **'\r'**, **'\t'**, **'\v'**) and the following table; these escape sequences shall be recognized both inside and outside bracket expressions. Note that records need not be separated by <newline>s and string constants can contain <newline>s, so even the **"\n"** sequence is valid in *awk* EREs. Using a slash character within an ERE requires the escaping shown in the following table.

Table: Escape Sequences in ***awk***

Escape		
Sequence	**Description**	**Meaning**
\"	*Backslash quotation-mark*	*Quotation-mark character*
\/	*Backslash slash*	*Slash character*
ddd	*A backslash character followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0 (that is, representation of the NUL character), the behavior is undefined.*	*The character whose encoding is represented by the one, two, or three-digit octal integer. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading* **'\'** *for each byte.*

\c	*A backslash character followed by any character not described in this table or in the table in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 5, File Format Notation (* **’\\\\’**, **’\a’**, **’\b’**, **’\f’**, **’\n’**, **’\r’**, **’\t’**, **’\v’** *).*	*Undefined*

A regular expression can be matched against a specific field or string by using one of the two regular expression matching operators, **’~’** and **"!~"** . These operators shall interpret their right-hand operand as a regular expression and their left-hand operand as a string. If the regular expression matches the string, the **’~’** expression shall evaluate to a value of 1, and the **"!~"** expression shall evaluate to a value of 0. (The regular expression matching operation is as defined by the term matched in the Base Definitions volume of IEEE Std 1003.1-2001, Section 9.1, Regular Expression Definitions, where a match occurs on any part of the string unless the regular expression is limited with the circumflex or dollar sign special characters.) If the regular expression does not match the string, the **’~’** expression shall evaluate to a value of 0, and the **"!~"** expression shall evaluate to a value of 1. If the right-hand operand is any expression other than the lexical token **ERE**, the string value of the expression shall be interpreted as an extended regular expression, including the escape conventions described above. Note that these same escape conventions shall also be applied in determining the value of a string literal (the lexical token **STRING**), and thus shall be applied a second time when a string literal is used in this context.

When an **ERE** token appears as an expression in any context other than as the right-hand of the **’~’** or **"!~"** operator or as one of the built-in function arguments described below, the value of the resulting expression shall be the equivalent of:

$0 ~ */ere/*

The *ere* argument to the **gsub**, **match**, **sub** functions, and the *fs* argument to the **split** function (see String Functions) shall be interpreted as extended regular expressions. These can be either **ERE** tokens or arbitrary expressions, and shall be interpreted in the same manner as the right-hand side of the **’~’** or **"!~"** operator.

An extended regular expression can be used to separate fields by using the **-F** *ERE* option or by assigning a string containing the expression to the built-in variable **FS**. The default value of the **FS** variable shall be a single <space>. The following describes **FS** behavior:

1. If **FS** is a null string, the behavior is unspecified.
2. If **FS** is a single character:
   a. If **FS** is <space>, skip leading and trailing <blank>s; fields shall be delimited by sets of one or more <blank>s.
   b. Otherwise, if **FS** is any other character *c*, fields shall be delimited by each single occurrence of *c*.
3. Otherwise, the string value of **FS** shall be considered to be an extended regular expression. Each occurrence of a sequence matching the extended regular expression shall delimit fields.

Except for the **’~’** and **"!~"** operators, and in the **gsub**, **match**, **split**, and **sub** built-in functions, ERE matching shall be based on input records; that is, record separator characters (the first character of the value of the variable **RS**, <newline> by default) cannot be embedded in the expression, and no expression shall match the record separator character. If the record separator is not <newline>, <newline>s embedded in the expression can be matched. For the **’~’** and **"!~"** operators, and in those four built-in functions, ERE matching shall be based on text strings; that is, any character (including <newline> and the record separator) can be embedded in the pattern, and an appropriate pattern shall match any character. However, in all *awk* ERE matching, the use of one or more NUL characters in the pattern, input record, or text string produces undefined results.

Patterns

A *pattern* is any valid *expression*, a range specified by two expressions separated by a comma, or one of the two special patterns **BEGIN** or **END**.

Special Patterns

The *awk* utility shall recognize two special patterns, **BEGIN** and **END**. Each **BEGIN** pattern shall be matched once and its associated action executed before the first record of input is read (except possibly by use of the **getline** function-see Input/Output and General Functions - in a prior **BEGIN** action) and before command line assignment is done. Each **END** pattern shall be matched once and its associated action executed after the last record of input has been read. These two patterns shall have associated actions.

BEGIN and **END** shall not combine with other patterns. Multiple **BEGIN** and **END** patterns shall be allowed. The actions associated with the **BEGIN** patterns shall be executed in the order specified in the program, as are the **END** actions. An **END** pattern can precede a **BEGIN** pattern in a program.

If an *awk* program consists of only actions with the pattern **BEGIN**, and the **BEGIN** action contains no **getline** function, *awk* shall exit without reading its input when the last statement in the last **BEGIN** action is executed. If an *awk* program consists of only actions with the pattern **END** or only actions with the patterns **BEGIN** and **END**, the input shall be read before the statements in the **END** actions are executed.

Expression Patterns

An expression pattern shall be evaluated as if it were an expression in a Boolean context. If the result is true, the pattern shall be considered to match, and the associated action (if any) shall be executed. If the result is false, the action shall not be executed.

Pattern Ranges

A pattern range consists of two expressions separated by a comma; in this case, the action shall be performed for all records between a match of the first expression and the following match of the second expression, inclusive. At this point, the pattern range can be repeated starting at input records subsequent to the end of the matched range.

Actions

An action is a sequence of statements as shown in the grammar in Grammar . Any single statement can be replaced by a statement list enclosed in braces. The application shall ensure that statements in a statement list are separated by <newline>s or semicolons. Statements in a statement list shall be executed sequentially in the order that they appear.

The *expression* acting as the conditional in an **if** statement shall be evaluated and if it is non-zero or non-null, the following statement shall be executed; otherwise, if **else** is present, the statement following the **else** shall be executed.

The **if**, **while**, **do**... **while**, **for**, **break**, and **continue** statements are based on the ISO C standard (see *Concepts Derived from the ISO C Standard*), except that the Boolean expressions shall be treated as described in Expressions in awk , and except in the case of:

```
for (variable in array)
```

which shall iterate, assigning each *index* of *array* to *variable* in an unspecified order. The results of adding new elements to *array* within such a **for** loop are undefined. If a **break** or **continue** statement occurs outside of a loop, the behavior is undefined.

The **delete** statement shall remove an individual array element. Thus, the following code deletes an entire array:

```
for (index in array)
  delete array[index]
```

The **next** statement shall cause all further processing of the current input record to be abandoned. The behavior is undefined if a **next** statement appears or is invoked in a **BEGIN** or **END** action.

The **exit** statement shall invoke all **END** actions in the order in which they occur in the program source and then terminate the program without reading further input. An **exit** statement inside an **END** action shall terminate the program without further execution of **END** actions. If an expression is specified in an **exit** statement, its numeric value shall be the exit status of *awk*, unless subsequent errors are encountered or a subsequent **exit** statement with an expression is executed.

Output Statements

Both **print** and **printf** statements shall write to standard output by default. The output shall be written to the location specified by *output_redirection* if one is supplied, as follows:

> *expression***>>** *expression***|** *expression*

In all cases, the *expression* shall be evaluated to produce a string that is used as a pathname into which to write (for **'>'** or **">>"**) or as a command to be executed (for **'|'**). Using the first two forms, if the file of that name is not currently open, it shall be opened, creating it if necessary and using the first form, truncating the file. The output then shall be appended to the file. As long as the file remains open, subsequent calls in which *expression* evaluates to the same string value shall simply append output to the file. The file remains open until the **close** function (see Input/Output and General Functions) is called with an expression that evaluates to the same string value.

The third form shall write output onto a stream piped to the input of a command. The stream shall be created if no stream is currently open with the value of *expression* as its command name. The stream created shall be equivalent to one created by a call to the *popen*() function defined in the System Interfaces volume of IEEE Std 1003.1-2001 with the value of *expression* as the *command* argument and a value of *w* as the *mode* argument. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value shall write output to the existing stream. The stream shall remain open until the **close** function (see Input/Output and General Functions) is called with an expression that evaluates to the same string value. At that time, the stream shall be closed as if by a call to the *pclose*() function defined in the System Interfaces volume of IEEE Std 1003.1-2001.

As described in detail by the grammar in Grammar , these output statements shall take a comma-separated list of *expression*s referred to in the grammar by the non-terminal symbols **expr_list**, **print_expr_list**, or **print_expr_list_opt**. This list is referred to here as the *expression list*, and each member is referred to as an *expression argument*.

The **print** statement shall write the value of each expression argument onto the indicated output stream separated by the current output field separator (see variable **OFS** above), and terminated by the output record separator (see variable **ORS** above). All expression arguments shall be taken as strings, being converted if necessary; this conversion shall be as described in Expressions in awk , with the exception that the **printf** format in **OFMT** shall be used instead of the value in **CONVFMT**. An empty expression list shall stand for the whole input record ($0).

The **printf** statement shall produce output based on a notation similar to the File Format Notation used to describe file formats in this volume of IEEE Std 1003.1-2001 (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 5, File Format Notation). Output shall be produced as specified with the first *expression* argument as the string *format* and subsequent *expression* arguments as the strings *arg1* to *argn*, inclusive, with the following exceptions:

1. The *format* shall be an actual character string rather than a graphical representation. Therefore, it cannot contain empty character positions. The <space> in the *format* string, in any context other than a *flag* of a conversion specification, shall be treated as an ordinary character that is copied to the output.

2. If the character set contains a **' '** character and that character appears in the *format* string, it shall be treated as an ordinary character that is copied to the output.

3. The *escape sequences* beginning with a backslash character shall be treated as sequences of ordinary characters that are copied to the output. Note that these same sequences shall be interpreted lexically by *awk* when they appear in literal strings, but they shall not be treated specially by the **printf** statement.

4. A *field width* or *precision* can be specified as the **'*'** character instead of a digit string. In this case the next argument from the expression list shall be fetched and its numeric value taken as the field width or precision.

5. The implementation shall not precede or follow output from the **d** or **u** conversion specifier characters with <blank>s not specified by the *format* string.

6. The implementation shall not precede output from the **o** conversion specifier character with leading zeros not specified by the *format* string.

7. For the **c** conversion specifier character: if the argument has a numeric value, the character whose encoding is that value shall be output. If the value is zero or is not the encoding of any character in the character set, the behavior is undefined. If the argument does not have a numeric value, the first character of the string value shall be output; if the string does not contain any characters, the behavior is undefined.

8. For each conversion specification that consumes an argument, the next expression argument shall be evaluated. With the exception of the **c** conversion specifier character, the value shall be converted (according to the rules specified in Expressions in awk) to the appropriate type for the conversion specification.

9. If there are insufficient expression arguments to satisfy all the conversion specifications in the *format* string, the behavior is undefined.

10. If any character sequence in the *format* string begins with a **'%'** character, but does not form a valid conversion specification, the behavior is unspecified.

Both **print** and **printf** can output at least {LINE_MAX} bytes.

Functions

The *awk* language has a variety of built-in functions: arithmetic, string, input/output, and general.

Arithmetic Functions

The arithmetic functions, except for **int**, shall be based on the ISO C standard (see *Concepts Derived from the ISO C Standard*). The behavior is undefined in cases where the ISO C standard specifies that an error be returned or that the behavior is undefined. Although the grammar (see Grammar) permits built-in functions to appear with no arguments or parentheses, unless the argument or parentheses are indicated as optional in the following list (by displaying them within the **"[]"** brackets), such use is undefined.

atan2(*y*,*x*)
Return arctangent of *y*/*x* in radians in the range [-pi,pi].

cos(*x*) Return cosine of *x*, where *x* is in radians.

sin(*x*) Return sine of *x*, where *x* is in radians.

exp(*x*) Return the exponential function of *x*.

log(*x*) Return the natural logarithm of *x*.

sqrt(*x*) Return the square root of *x*.

int(*x*) Return the argument truncated to an integer. Truncation shall be toward 0 when *x*>0.

rand() Return a random number *n*, such that 0<=*n*<1.

srand([*expr*])
Set the seed value for *rand* to *expr* or use the time of day if *expr* is omitted. The previous seed value shall be returned.

String Functions

The string functions in the following list shall be supported. Although the grammar (see Grammar) permits built-in functions to appear with no arguments or parentheses, unless the argument or parentheses are indicated as optional in the following list (by displaying them within the **"[]"** brackets), such use is undefined.

gsub(*ere*, *repl*[, *in*])
Behave like **sub** (see below), except that it shall replace all occurrences of the regular expression (like the *ed* utility global substitute) in $0 or in the *in* argument, when specified.

index(*s*, *t*)
: Return the position, in characters, numbering from 1, in string *s* where string *t* first occurs, or zero if it does not occur at all.

length[([*s*])]
: Return the length, in characters, of its argument taken as a string, or of the whole record, $0, if there is no argument.

match(*s*, *ere*)
: Return the position, in characters, numbering from 1, in string *s* where the extended regular expression *ere* occurs, or zero if it does not occur at all. RSTART shall be set to the starting position (which is the same as the returned value), zero if no match is found; RLENGTH shall be set to the length of the matched string, -1 if no match is found.

split(*s*, *a*[, *fs*])
: Split the string *s* into array elements *a*[1], *a*[2], ..., *a*[*n*], and return *n*. All elements of the array shall be deleted before the split is performed. The separation shall be done with the ERE *fs* or with the field separator **FS** if *fs* is not given. Each array element shall have a string value when created and, if appropriate, the array element shall be considered a numeric string (see Expressions in awk). The effect of a null string as the value of *fs* is unspecified.

sprintf(*fmt*, *expr*, *expr*, ...)
: Format the expressions according to the **printf** format given by *fmt* and return the resulting string.

sub(*ere*, *repl*[, *in*])
: Substitute the string *repl* in place of the first instance of the extended regular expression *ERE* in string *in* and return the number of substitutions. An ampersand (**'&'**) appearing in the string *repl* shall be replaced by the string from *in* that matches the ERE. An ampersand preceded with a backslash (**'\'**) shall be interpreted as the literal ampersand character. An occurrence of two consecutive backslashes shall be interpreted as just a single literal backslash character. Any other occurrence of a backslash (for example, preceding any other character) shall be treated as a literal backslash character. Note that if *repl* is a string literal (the lexical token **STRING**; see Grammar), the handling of the ampersand character occurs after any lexical processing, including any lexical backslash escape sequence processing. If *in* is specified and it is not an lvalue (see Expressions in awk), the behavior is undefined. If *in* is omitted, *awk* shall use the current record ($0) in its place.

substr(*s*, *m*[, *n*])
: Return the at most *n*-character substring of *s* that begins at position *m*, numbering from 1. If *n* is omitted, or if *n* specifies more characters than are left in the string, the length of the substring shall be limited by the length of the string *s*.

tolower(*s*)
: Return a string based on the string *s*. Each character in *s* that is an uppercase letter specified to have a **tolower** mapping by the *LC_CTYPE* category of the current locale shall be replaced in the returned string by the lowercase letter specified by the mapping. Other characters in *s* shall be unchanged in the returned string.

toupper(*s*)
: Return a string based on the string *s*. Each character in *s* that is a lowercase letter specified to have a **toupper** mapping by the *LC_CTYPE* category of the current locale is replaced in the returned string by the uppercase letter specified by the mapping. Other characters in *s* are unchanged in the returned string.

All of the preceding functions that take *ERE* as a parameter expect a pattern or a string valued expression that is a regular expression as defined in Regular Expressions .

Input/Output and General Functions

The input/output and general functions are:

close(*expression*)
: Close the file or pipe opened by a **print** or **printf** statement or a call to **getline** with the same string-valued *expression*. The limit on the number of open *expression* arguments is

implementation-defined. If the close was successful, the function shall return zero; otherwise, it shall return non-zero.

expression | **getline [***var***]**
: Read a record of input from a stream piped from the output of a command. The stream shall be created if no stream is currently open with the value of *expression* as its command name. The stream created shall be equivalent to one created by a call to the *popen*() function with the value of *expression* as the *command* argument and a value of *r* as the *mode* argument. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value shall read subsequent records from the stream. The stream shall remain open until the **close** function is called with an expression that evaluates to the same string value. At that time, the stream shall be closed as if by a call to the *pclose*() function. If *var* is omitted, $0 and **NF** shall be set; otherwise, *var* shall be set and, if appropriate, it shall be considered a numeric string (see Expressions in awk).

The **getline** operator can form ambiguous constructs when there are unparenthesized operators (including concatenate) to the left of the **'|'** (to the beginning of the expression containing **getline**). In the context of the **'$'** operator, **'|'** shall behave as if it had a lower precedence than **'$'** . The result of evaluating other operators is unspecified, and conforming applications shall parenthesize properly all such usages.

getline Set $0 to the next input record from the current input file. This form of **getline** shall set the **NF**, **NR**, and **FNR** variables.

getline *var*
: Set variable *var* to the next input record from the current input file and, if appropriate, *var* shall be considered a numeric string (see Expressions in awk). This form of **getline** shall set the **FNR** and **NR** variables.

getline [*var***]** < *expression*
: Read the next record of input from a named file. The *expression* shall be evaluated to produce a string that is used as a pathname. If the file of that name is not currently open, it shall be opened. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value shall read subsequent records from the file. The file shall remain open until the **close** function is called with an expression that evaluates to the same string value. If *var* is omitted, $0 and **NF** shall be set; otherwise, *var* shall be set and, if appropriate, it shall be considered a numeric string (see Expressions in awk).

The **getline** operator can form ambiguous constructs when there are unparenthesized binary operators (including concatenate) to the right of the **'<'** (up to the end of the expression containing the **getline**). The result of evaluating such a construct is unspecified, and conforming applications shall parenthesize properly all such usages.

system(*expression*)
: Execute the command given by *expression* in a manner equivalent to the *system*() function defined in the System Interfaces volume of IEEE Std 1003.1-2001 and return the exit status of the command.

All forms of **getline** shall return 1 for successful input, zero for end-of-file, and -1 for an error.

Where strings are used as the name of a file or pipeline, the application shall ensure that the strings are textually identical. The terminology "same string value" implies that "equivalent strings", even those that differ only by <space>s, represent different files.

User-Defined Functions

The *awk* language also provides user-defined functions. Such functions can be defined as:

function *name*(**[***parameter***, ...]**) **{** *statements* **}**

A function can be referred to anywhere in an *awk* program; in particular, its use can precede its definition. The scope of a function is global.

Function parameters, if present, can be either scalars or arrays; the behavior is undefined if an array name is passed as a parameter that the function uses as a scalar, or if a scalar expression is passed as a parameter that the function uses as an array. Function parameters shall be passed by value if scalar and by reference if array name.

The number of parameters in the function definition need not match the number of parameters in the function call. Excess formal parameters can be used as local variables. If fewer arguments are supplied in a function call than are in the function definition, the extra parameters that are used in the function body as scalars shall evaluate to the uninitialized value until they are otherwise initialized, and the extra parameters that are used in the function body as arrays shall be treated as uninitialized arrays where each element evaluates to the uninitialized value until otherwise initialized.

When invoking a function, no white space can be placed between the function name and the opening parenthesis. Function calls can be nested and recursive calls can be made upon functions. Upon return from any nested or recursive function call, the values of all of the calling function's parameters shall be unchanged, except for array parameters passed by reference. The **return** statement can be used to return a value. If a **return** statement appears outside of a function definition, the behavior is undefined.

In the function definition, <newline>s shall be optional before the opening brace and after the closing brace. Function definitions can appear anywhere in the program where a *pattern-action* pair is allowed.

Grammar

The grammar in this section and the lexical conventions in the following section shall together describe the syntax for *awk* programs. The general conventions for this style of grammar are described in *Grammar Conventions* . A valid program can be represented as the non-terminal symbol *program* in the grammar. This formal syntax shall take precedence over the preceding text syntax description.

```
%token NAME NUMBER STRING ERE
%token FUNC_NAME   /* Name followed by '(' without white space. */


/* Keywords  */
%token      Begin   End
/*          'BEGIN' 'END'                    */


%token      Break   Continue   Delete   Do   Else
/*          'break' 'continue' 'delete' 'do' 'else'  */


%token      Exit   For   Function   If   In
/*          'exit' 'for' 'function' 'if' 'in'       */


%token      Next   Print   Printf   Return   While
/*          'next' 'print' 'printf' 'return' 'while' */


/* Reserved function names */
%token BUILTIN_FUNC_NAME
            /* One token for the following:
             * atan2 cos sin exp log sqrt int rand srand
             * gsub index length match split sprintf sub
             * substr tolower toupper close system
             */
%token GETLINE
            /* Syntactically different from other built-ins. */


/* Two-character tokens. */
%token ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN POW_ASSIGN
/*     '+='       '-='       '*='       '/='       '%='       '^=' */
```

```
%token OR   AND  NO_MATCH   EQ   LE   GE   NE   INCR  DECR  APPEND
/*     '||'  '&&'  '!~'  '=='  '<='  '>='  '!='  '++'   '--'   '>>'    */


/* One-character tokens. */
%token '{' '}' '(' ')' '[' ']' ',' ';' NEWLINE
%token '+' '-' '*' '%' '^' '!' '>' '<' '|' '?' ':' '~' '$' '='


%start program
%%


program          : item_list
                 | actionless_item_list
                 ;


item_list        : newline_opt
                 | actionless_item_list item terminator
                 | item_list            item terminator
                 | item_list          action terminator
                 ;


actionless_item_list : item_list            pattern terminator
                 | actionless_item_list pattern terminator
                 ;


item             : pattern action
                 | Function NAME      '(' param_list_opt ')'
                      newline_opt action
                 | Function FUNC_NAME '(' param_list_opt ')'
                      newline_opt action
                 ;


param_list_opt   : /* empty */
                 | param_list
                 ;


param_list       : NAME
                 | param_list ',' NAME
                 ;


pattern          : Begin
                 | End
                 | expr
                 | expr ',' newline_opt expr
                 ;


action           : '{' newline_opt                             '}'
                 | '{' newline_opt terminated_statement_list   '}'
                 | '{' newline_opt unterminated_statement_list '}'
                 ;
```

```
terminator       : terminator ';'
             | terminator NEWLINE
             |          ';'
             |          NEWLINE
             ;


terminated_statement_list : terminated_statement
             | terminated_statement_list terminated_statement
             ;


unterminated_statement_list : unterminated_statement
             | terminated_statement_list unterminated_statement
             ;


terminated_statement : action newline_opt
             | If '(' expr ')' newline_opt terminated_statement
             | If '(' expr ')' newline_opt terminated_statement
                  Else newline_opt terminated_statement
             | While '(' expr ')' newline_opt terminated_statement
             | For '(' simple_statement_opt ';'
                 expr_opt ';' simple_statement_opt ')' newline_opt
                 terminated_statement
             | For '(' NAME In NAME ')' newline_opt
                 terminated_statement
             | ';' newline_opt
             | terminatable_statement NEWLINE newline_opt
             | terminatable_statement ';'     newline_opt
             ;


unterminated_statement : terminatable_statement
             | If '(' expr ')' newline_opt unterminated_statement
             | If '(' expr ')' newline_opt terminated_statement
                 Else newline_opt unterminated_statement
             | While '(' expr ')' newline_opt unterminated_statement
             | For '(' simple_statement_opt ';'
              expr_opt ';' simple_statement_opt ')' newline_opt
                 unterminated_statement
             | For '(' NAME In NAME ')' newline_opt
                 unterminated_statement
             ;


terminatable_statement : simple_statement
             | Break
             | Continue
             | Next
             | Exit expr_opt
             | Return expr_opt
             | Do newline_opt terminated_statement While '(' expr ')'
             ;


simple_statement_opt : /* empty */
             | simple_statement
             ;
```

```
simple_statement : Delete NAME '[' expr_list ']'
           | expr
           | print_statement
           ;


print_statement  : simple_print_statement
           | simple_print_statement output_redirection
           ;


simple_print_statement : Print  print_expr_list_opt
           | Print  '(' multiple_expr_list ')'
           | Printf print_expr_list
           | Printf '(' multiple_expr_list ')'
           ;


output_redirection : '>'    expr
           | APPEND expr
           | '|'    expr
           ;


expr_list_opt    : /* empty */
           | expr_list
           ;


expr_list        : expr
           | multiple_expr_list
           ;


multiple_expr_list : expr ',' newline_opt expr
           | multiple_expr_list ',' newline_opt expr
           ;


expr_opt         : /* empty */
           | expr
           ;


expr             : unary_expr
           | non_unary_expr
           ;


unary_expr       : '+' expr
           | '-' expr
           | unary_expr '^'      expr
           | unary_expr '*'      expr
           | unary_expr '/'      expr
           | unary_expr '%'      expr
           | unary_expr '+'      expr
           | unary_expr '-'      expr
           | unary_expr          non_unary_expr
           | unary_expr '<'      expr
```

```
                 | unary_expr LE      expr
                 | unary_expr NE      expr
                 | unary_expr EQ      expr
                 | unary_expr '>'     expr
                 | unary_expr GE      expr
                 | unary_expr '˜'     expr
                 | unary_expr NO_MATCH expr
                 | unary_expr In NAME
                 | unary_expr AND newline_opt expr
                 | unary_expr OR  newline_opt expr
                 | unary_expr '?' expr ':' expr
                 | unary_input_function
                 ;


non_unary_expr   : '(' expr ')'
                 | '!' expr
                 | non_unary_expr '^'     expr
                 | non_unary_expr '*'     expr
                 | non_unary_expr '/'     expr
                 | non_unary_expr '%'     expr
                 | non_unary_expr '+'     expr
                 | non_unary_expr '-'     expr
                 | non_unary_expr         non_unary_expr
                 | non_unary_expr '<'     expr
                 | non_unary_expr LE      expr
                 | non_unary_expr NE      expr
                 | non_unary_expr EQ      expr
                 | non_unary_expr '>'     expr
                 | non_unary_expr GE      expr
                 | non_unary_expr '˜'     expr
                 | non_unary_expr NO_MATCH expr
                 | non_unary_expr In NAME
                 | '(' multiple_expr_list ')' In NAME
                 | non_unary_expr AND newline_opt expr
                 | non_unary_expr OR  newline_opt expr
                 | non_unary_expr '?' expr ':' expr
                 | NUMBER
                 | STRING
                 | lvalue
                 | ERE
                 | lvalue INCR
                 | lvalue DECR
                 | INCR lvalue
                 | DECR lvalue
                 | lvalue POW_ASSIGN expr
                 | lvalue MOD_ASSIGN expr
                 | lvalue MUL_ASSIGN expr
                 | lvalue DIV_ASSIGN expr
                 | lvalue ADD_ASSIGN expr
                 | lvalue SUB_ASSIGN expr
                 | lvalue '=' expr
                 | FUNC_NAME '(' expr_list_opt ')'
                     /* no white space allowed before '(' */
                 | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
                 | BUILTIN_FUNC_NAME
                 | non_unary_input_function
                 ;
```

```
print_expr_list_opt : /* empty */
             | print_expr_list
             ;


print_expr_list  : print_expr
             | print_expr_list ’,’ newline_opt print_expr
             ;


print_expr       : unary_print_expr
             | non_unary_print_expr
             ;


unary_print_expr : ’+’ print_expr
             | ’-’ print_expr
             | unary_print_expr ’^’      print_expr
             | unary_print_expr ’*’      print_expr
             | unary_print_expr ’/’      print_expr
             | unary_print_expr ’%’      print_expr
             | unary_print_expr ’+’      print_expr
             | unary_print_expr ’-’      print_expr
             | unary_print_expr          non_unary_print_expr
             | unary_print_expr ’~’      print_expr
             | unary_print_expr NO_MATCH print_expr
             | unary_print_expr In NAME
             | unary_print_expr AND newline_opt print_expr
             | unary_print_expr OR  newline_opt print_expr
             | unary_print_expr ’?’ print_expr ’:’ print_expr
             ;


non_unary_print_expr : ’(’ expr ’)’
             | ’!’ print_expr
             | non_unary_print_expr ’^’      print_expr
             | non_unary_print_expr ’*’      print_expr
             | non_unary_print_expr ’/’      print_expr
             | non_unary_print_expr ’%’      print_expr
             | non_unary_print_expr ’+’      print_expr
             | non_unary_print_expr ’-’      print_expr
             | non_unary_print_expr          non_unary_print_expr
             | non_unary_print_expr ’~’      print_expr
             | non_unary_print_expr NO_MATCH print_expr
             | non_unary_print_expr In NAME
             | ’(’ multiple_expr_list ’)’ In NAME
             | non_unary_print_expr AND newline_opt print_expr
             | non_unary_print_expr OR  newline_opt print_expr
             | non_unary_print_expr ’?’ print_expr ’:’ print_expr
             | NUMBER
             | STRING
             | lvalue
             | ERE
             | lvalue INCR
             | lvalue DECR
             | INCR lvalue
             | DECR lvalue
             | lvalue POW_ASSIGN print_expr
             | lvalue MOD_ASSIGN print_expr
```

```
                 | lvalue MUL_ASSIGN print_expr
                 | lvalue DIV_ASSIGN print_expr
                 | lvalue ADD_ASSIGN print_expr
                 | lvalue SUB_ASSIGN print_expr
                 | lvalue ’=’ print_expr
                 | FUNC_NAME ’(’ expr_list_opt ’)’
                    /* no white space allowed before ’(’ */
                 | BUILTIN_FUNC_NAME ’(’ expr_list_opt ’)’
                 | BUILTIN_FUNC_NAME
                 ;


lvalue           : NAME
                 | NAME ’[’ expr_list ’]’
                 | ’$’ expr
                 ;


non_unary_input_function : simple_get
                 | simple_get ’<’ expr
                 | non_unary_expr ’|’ simple_get
                 ;


unary_input_function : unary_expr ’|’ simple_get
                 ;


simple_get       : GETLINE
                 | GETLINE lvalue
                 ;


newline_opt      : /* empty */
                 | newline_opt NEWLINE
                 ;
```

This grammar has several ambiguities that shall be resolved as follows:

* Operator precedence and associativity shall be as described in Expressions in Decreasing Precedence in *awk* .

* In case of ambiguity, an **else** shall be associated with the most immediately preceding **if** that would satisfy the grammar.

* In some contexts, a slash (**’/’**) that is used to surround an ERE could also be the division operator. This shall be resolved in such a way that wherever the division operator could appear, a slash is assumed to be the division operator. (There is no unary division operator.)

One convention that might not be obvious from the formal grammar is where <newline>s are acceptable. There are several obvious placements such as terminating a statement, and a backslash can be used to escape <newline>s between any lexical tokens. In addition, <newline>s without backslashes can follow a comma, an open brace, logical AND operator (**"&&"**), logical OR operator (**"||"**), the **do** keyword, the **else** keyword, and the closing parenthesis of an **if**, **for**, or **while** statement. For example:

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

Lexical Conventions

The lexical conventions for *awk* programs, with respect to the preceding grammar, shall be as follows:

1. Except as noted, *awk* shall recognize the longest possible token or delimiter beginning at a given point.

2. A comment shall consist of any characters beginning with the number sign character and terminated by, but excluding the next occurrence of, a <newline>. Comments shall have no effect, except to delimit lexical tokens.

3. The <newline> shall be recognized as the token **NEWLINE**.

4. A backslash character immediately followed by a <newline> shall have no effect.

5. The token **STRING** shall represent a string constant. A string constant shall begin with the character **' .'** Within a string constant, a backslash character shall be considered to begin an escape sequence as specified in the table in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 5, File Format Notation (**'\\'**, **'\a'**, **'\b'**, **'\f'**, **'\n'**, **'\r'**, **'\t'**, **'\v'**). In addition, the escape sequences in Expressions in Decreasing Precedence in *awk* shall be recognized. A <newline> shall not occur within a string constant. A string constant shall be terminated by the first unescaped occurrence of the character **''** after the one that begins the string constant. The value of the string shall be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting **''** characters.

6. The token **ERE** represents an extended regular expression constant. An ERE constant shall begin with the slash character. Within an ERE constant, a backslash character shall be considered to begin an escape sequence as specified in the table in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 5, File Format Notation. In addition, the escape sequences in Expressions in Decreasing Precedence in *awk* shall be recognized. The application shall ensure that a <newline> does not occur within an ERE constant. An ERE constant shall be terminated by the first unescaped occurrence of the slash character after the one that begins the ERE constant. The extended regular expression represented by the ERE constant shall be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting slash characters.

7. A <blank> shall have no effect, except to delimit lexical tokens or within **STRING** or **ERE** tokens.

8. The token **NUMBER** shall represent a numeric constant. Its form and numeric value shall be equivalent to either of the tokens **floating-constant** or **integer-constant** as specified by the ISO C standard, with the following exceptions:

   a. An integer constant cannot begin with 0x or include the hexadecimal digits **'a'**, **'b'**, **'c'**, **'d'**, **'e'**, **'f'**, **'A'**, **'B'**, **'C'**, **'D'**, **'E'**, or **'F'** .

   b. The value of an integer constant beginning with 0 shall be taken in decimal rather than octal.

   c. An integer constant cannot include a suffix (**'u'**, **'U'**, **'l'**, or **'L'**).

   d. A floating constant cannot include a suffix (**'f'**, **'F'**, **'l'**, or **'L'**).

   If the value is too large or too small to be representable (see *Concepts Derived from the ISO C Standard*), the behavior is undefined.

9. A sequence of underscores, digits, and alphabetics from the portable character set (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 6.1, Portable Character Set), beginning with an underscore or alphabetic, shall be considered a word.

10. The following words are keywords that shall be recognized as individual tokens; the name of the token is the same as the keyword:

BEGIN	**delete**	**END**	**function**	**in**	**printf**
break	**do**	**exit**	**getline**	**next**	**return**
continue	**else**	**for**	**if**	**print**	**while**

11. The following words are names of built-in functions and shall be recognized as the token **BUILTIN_FUNC_NAME**:

atan2	**gsub**	**log**	**split**	**sub**	**toupper**
close	**index**	**match**	**sprintf**	**substr**	
cos	**int**	**rand**	**sqrt**	**system**	
exp	**length**	**sin**	**srand**	**tolower**	

The above-listed keywords and names of built-in functions are considered reserved words.

12. The token **NAME** shall consist of a word that is not a keyword or a name of a built-in function and is not followed immediately (without any delimiters) by the **'('** character.

13. The token **FUNC_NAME** shall consist of a word that is not a keyword or a name of a built-in function, followed immediately (without any delimiters) by the **'('** character. The **'('** character shall not be included as part of the token.

14. The following two-character sequences shall be recognized as the named tokens:

Token Name	**Sequence**	**Token Name**	**Sequence**
ADD_ASSIGN	+=	**NO_MATCH**	!˜
SUB_ASSIGN	-=	**EQ**	==
MUL_ASSIGN	*=	**LE**	<=
DIV_ASSIGN	/=	**GE**	>=
MOD_ASSIGN	%=	**NE**	!=
POW_ASSIGN	ˆ=	**INCR**	++
OR	||	**DECR**	--
AND	&&	**APPEND**	>>

15. The following single characters shall be recognized as tokens whose names are the character:

<newline> { } ( ) [ ] , ; + - * % ˆ ! > < | ? : ˜ $ =

There is a lexical ambiguity between the token **ERE** and the tokens **'/'** and **DIV_ASSIGN**. When an input sequence begins with a slash character in any syntactic context where the token **'/'** or **DIV_ASSIGN** could appear as the next token in a valid program, the longer of those two tokens that can be recognized shall be recognized. In any other syntactic context where the token **ERE** could appear as the next token in a valid program, the token **ERE** shall be recognized.

EXIT STATUS

The following exit values shall be returned:

0	All input files were processed successfully.
>0	An error occurred.

The exit status can be altered within the program by using an **exit** expression.

CONSEQUENCES OF ERRORS

If any *file* operand is specified and the named file cannot be accessed, *awk* shall write a diagnostic message to standard error and terminate without any further action.

If the program specified by either the *program* operand or a *progfile* operand is not a valid *awk* program (as specified in the EXTENDED DESCRIPTION section), the behavior is undefined.

The following sections are informative.

APPLICATION USAGE

The **index**, **length**, **match**, and **substr** functions should not be confused with similar functions in the ISO C standard; the *awk* versions deal with characters, while the ISO C standard deals with bytes.

Because the concatenation operation is represented by adjacent expressions rather than an explicit operator, it is often necessary to use parentheses to enforce the proper evaluation precedence.

EXAMPLES

The *awk* program specified in the command line is most easily specified within single-quotes (for example, programs commonly contain characters that are special to the shell, including double-quotes. In the cases where an *awk* program contains single-quote characters, it is usually easiest to specify most of the program as strings within single-quotes concatenated by the shell with quoted single-quote characters. For example:

```
awk ’/’\”/ { print "quote:", $0 }’
```

prints all lines from the standard input containing a single-quote character, prefixed with *quote*:.

The following are examples of simple *awk* programs:

1. Write to the standard output all input lines for which field 3 is greater than 5:

   ```
   $3 > 5
   ```

2. Write every tenth line:

   ```
   (NR % 10) == 0
   ```

3. Write any line with a substring matching the regular expression:

   ```
   /(G|D)(2[0-9][[:alpha:]]*)/
   ```

4. Print any line with a substring containing a **’G’** or **’D’**, followed by a sequence of digits and characters. This example uses character classes **digit** and **alpha** to match language-independent digit and alphabetic characters respectively:

   ```
   /(G|D)([[:digit:][:alpha:]]*)/
   ```

5. Write any line in which the second field matches the regular expression and the fourth field does not:

   ```
   $2 ˜ /xyz/ && $4 !˜ /xyz/
   ```

6. Write any line in which the second field contains a backslash:

   ```
   $2 ˜ /\\/
   ```

7. Write any line in which the second field contains a backslash. Note that backslash escapes are interpreted twice; once in lexical processing of the string and once in processing the regular expression:

   ```
   $2 ˜ "\\\\"
   ```

8. Write the second to the last and the last field in each line. Separate the fields by a colon:

```
{OFS=":";print $(NF-1), $NF}
```

9. Write the line number and number of fields in each line. The three strings representing the line number, the colon, and the number of fields are concatenated and that string is written to standard output:

```
{print NR ":" NF}
```

10. Write lines longer than 72 characters:

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

11. Write the first two fields in opposite order separated by **OFS**:

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

12. Same, with input fields separated by a comma or <space>s and <tab>s, or both:

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

13. Add up the first column, print sum, and average:

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

14. Write fields in reverse order, one per line (many lines out for each line in):

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

15. Write all lines between occurrences of the strings **start** and **stop**:

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

16. Write all lines whose first field is different from the previous one:

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

17. Simulate *echo*:

```
BEGIN {
    for (i = 1; i < ARGC; ++i)
    printf("%s%s", ARGV[i], i==ARGC-1?"\n":" ")
}
```

18. Write the path prefixes contained in the *PATH* environment variable, one per line:

```
BEGIN {
    n = split (ENVIRON["PATH"], path, ":")
    for (i = 1; i <= n; ++i)
    print path[i]
}
```

19. If there is a file named **input** containing page headers of the form:

```
Page #
```

and a file named **program** that contains:

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

then the command line:

```
awk -f program n=5 input
```

prints the file **input**, filling in page numbers starting at 5.

RATIONALE

This description is based on the new *awk*, "nawk", (see the referenced *The AWK Programming Language*), which introduced a number of new features to the historical *awk*:

1. New keywords: **delete**, **do**, **function**, **return**
2. New built-in functions: **atan2**, **close**, **cos**, **gsub**, **match**, **rand**, **sin**, **srand**, **sub**, **system**
3. New predefined variables: **FNR**, **ARGC**, **ARGV**, **RSTART**, **RLENGTH**, **SUBSEP**
4. New expression operators: **?**, **:**, **,**, **ˆ**
5. The **FS** variable and the third argument to **split**, now treated as extended regular expressions.
6. The operator precedence, changed to more closely match the C language. Two examples of code that operate differently are:

```
while ( n /= 10 > 1) ...
if (!"wk" ˜ /bwk/) ...
```

Several features have been added based on newer implementations of *awk*:

* Multiple instances of **-f** *progfile* are permitted.
* The new option **-v** *assignment.*
* The new predefined variable **ENVIRON**.
* New built-in functions **toupper** and **tolower**.
* More formatting capabilities are added to **printf** to match the ISO C standard.

The overall *awk* syntax has always been based on the C language, with a few features from the shell command language and other sources. Because of this, it is not completely compatible with any other language, which has caused confusion for some users. It is not the intent of the standard developers to address such issues. A few relatively minor changes toward making the language more compatible with the ISO C standard were made; most of these changes are based on similar changes in recent implementations, as described above. There remain several C-language conventions that are not in *awk*. One of the notable ones is the comma operator, which is commonly used to specify multiple expressions in the C language **for** statement. Also, there are various places where *awk* is more restrictive than the C language regarding the type of expression that can be used in a given context. These limitations are due to the different features that the *awk* language does provide.

Regular expressions in *awk* have been extended somewhat from historical implementations to make them a pure superset of extended regular expressions, as defined by IEEE Std 1003.1-2001 (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 9.4, Extended Regular Expressions). The main extensions are internationalization features and interval expressions. Historical implementations of *awk* have long supported backslash escape sequences as an extension to extended regular expressions, and this extension has been retained despite inconsistency with other utilities. The number of escape sequences recognized in both extended regular expressions and strings has varied (generally increasing with time) among implementations. The set specified by IEEE Std 1003.1-2001 includes most sequences known to be supported by popular implementations and by the ISO C standard. One sequence that is not supported is hexadecimal value escapes beginning with **'\x'** . This would allow values expressed in more than 9 bits to be used within *awk* as in the ISO C standard. However, because this syntax has a non-deterministic length, it does not permit the subsequent character to be a hexadecimal digit. This limitation can be dealt with in the C language by the use of lexical string concatenation. In the *awk* language, concatenation could also be a solution for strings, but not for extended regular expressions (either lexical ERE tokens or strings used dynamically as regular expressions). Because of this limitation, the feature has not been added to IEEE Std 1003.1-2001.

When a string variable is used in a context where an extended regular expression normally appears (where the lexical token ERE is used in the grammar) the string does not contain the literal slashes.

Some versions of *awk* allow the form:

```
func name(args, ... ) { statements }
```

This has been deprecated by the authors of the language, who asked that it not be specified.

Historical implementations of *awk* produce an error if a **next** statement is executed in a **BEGIN** action, and cause *awk* to terminate if a **next** statement is executed in an **END** action. This behavior has not been documented, and it was not believed that it was necessary to standardize it.

The specification of conversions between string and numeric values is much more detailed than in the documentation of historical implementations or in the referenced *The AWK Programming Language*. Although most of the behavior is designed to be intuitive, the details are necessary to ensure compatible behavior from different implementations. This is especially important in relational expressions since the types of the operands determine whether a string or numeric comparison is performed. From the perspective of an application writer, it is usually sufficient to expect intuitive behavior and to force conversions (by adding zero or concatenating a null string) when the type of an expression does not obviously match what is needed. The intent has been to specify historical practice in almost all cases. The one exception is that, in historical implementations, variables and constants maintain both string and numeric values after their original value is converted by any use. This means that referencing a variable or constant can have unexpected side effects. For example, with historical implementations the following program:

```
{
  a = "+2"
  b = 2
  if (NR % 2)
    c = a + b
  if (a == b)
    print "numeric comparison"
  else
    print "string comparison"
}
```

would perform a numeric comparison (and output numeric comparison) for each odd-numbered line, but perform a string comparison (and output string comparison) for each even-numbered line. IEEE Std 1003.1-2001 ensures that comparisons will be numeric if necessary. With historical implementations, the following program:

```
BEGIN {
        OFMT = "%e"
        print 3.14
        OFMT = "%f"
        print 3.14
    }
```

would output **"3.140000e+00"** twice, because in the second **print** statement the constant **"3.14"** would have a string value from the previous conversion. IEEE Std 1003.1-2001 requires that the output of the second **print** statement be **"3.140000"** . The behavior of historical implementations was seen as too unintuitive and unpredictable.

It was pointed out that with the rules contained in early drafts, the following script would print nothing:

```
    BEGIN {
        y[1.5] = 1
        OFMT = "%e"
        print y[1.5]
    }
```

Therefore, a new variable, **CONVFMT**, was introduced. The **OFMT** variable is now restricted to affecting output conversions of numbers to strings and **CONVFMT** is used for internal conversions, such as comparisons or array indexing. The default value is the same as that for **OFMT**, so unless a program changes **CONVFMT** (which no historical program would do), it will receive the historical behavior associated with internal string conversions.

The POSIX *awk* lexical and syntactic conventions are specified more formally than in other sources. Again the intent has been to specify historical practice. One convention that may not be obvious from the formal grammar as in other verbal descriptions is where <newline>s are acceptable. There are several obvious placements such as terminating a statement, and a backslash can be used to escape <newline>s between any lexical tokens. In addition, <newline>s without backslashes can follow a comma, an open brace, a logical AND operator (**"&&"**), a logical OR operator (**"||"**), the **do** keyword, the **else** keyword, and the closing parenthesis of an **if**, **for**, or **while** statement. For example:

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

The requirement that *awk* add a trailing <newline> to the program argument text is to simplify the grammar, making it match a text file in form. There is no way for an application or test suite to determine whether a literal <newline> is added or whether *awk* simply acts as if it did.

IEEE Std 1003.1-2001 requires several changes from historical implementations in order to support internationalization. Probably the most subtle of these is the use of the decimal-point character, defined by the *LC_NUMERIC* category of the locale, in representations of floating-point numbers. This locale-specific character is used in recognizing numeric input, in converting between strings and numeric values, and in formatting output. However, 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). This is essentially the same convention as the one used in the ISO C standard. The difference is that the C language includes the *setlocale*() function, which permits an application to modify its locale. Because of this capability, a C application begins executing with its locale set to the C locale, and only executes in the environment-specified locale after an explicit call to *setlocale*(). However, adding such an elaborate new feature to the *awk* language was seen as inappropriate for IEEE Std 1003.1-2001. It is possible to execute an *awk* program explicitly in any desired locale by setting the environment in the shell.

The undefined behavior resulting from NULs in extended regular expressions allows future extensions for the GNU *gawk* program to process binary data.

The behavior in the case of invalid *awk* programs (including lexical, syntactic, and semantic errors) is undefined because it was considered overly limiting on implementations to specify. In most cases such errors can be expected to produce a diagnostic and a non-zero exit status. However, some implementations may choose to extend the language in ways that make use of certain invalid constructs. Other invalid constructs might be deemed worthy of a warning, but otherwise cause some reasonable behavior. Still other constructs may be very difficult to detect in some implementations. Also, different implementations might detect a given error during an initial parsing of the program (before reading any input files) while others might detect it when executing the program after reading some input. Implementors should be aware that diagnosing errors as early as possible and producing useful diagnostics can ease debugging of applications, and thus make an implementation more usable.

The unspecified behavior from using multi-character **RS** values is to allow possible future extensions based on extended regular expressions used for record separators. Historical implementations take the first character of the string and ignore the others.

Unspecified behavior when *split*(*string*, *array*, <null>) is used is to allow a proposed future extension that would split up a string into an array of individual characters.

In the context of the **getline** function, equally good arguments for different precedences of the **|** and **<** operators can be made. Historical practice has been that:

```
getline < "a" "b"
```

is parsed as:

```
( getline < "a" ) "b"
```

although many would argue that the intent was that the file **ab** should be read. However:

```
getline < "x" + 1
```

parses as:

```
getline < ( "x" + 1 )
```

Similar problems occur with the **|** version of **getline**, particularly in combination with **$**. For example:

```
$"echo hi" | getline
```

(This situation is particularly problematic when used in a **print** statement, where the **|getline** part might be a redirection of the **print**.)

Since in most cases such constructs are not (or at least should not) be used (because they have a natural ambiguity for which there is no conventional parsing), the meaning of these constructs has been made explicitly unspecified. (The effect is that a conforming application that runs into the problem must parenthesize to resolve the ambiguity.) There appeared to be few if any actual uses of such constructs.

Grammars can be written that would cause an error under these circumstances. Where backwards-compatibility is not a large consideration, implementors may wish to use such grammars.

Some historical implementations have allowed some built-in functions to be called without an argument list, the result being a default argument list chosen in some "reasonable" way. Use of **length** as a synonym for **length($0)** is the only one of these forms that is thought to be widely known or widely used; this particular form is documented in various places (for example, most historical *awk* reference pages, although not in the referenced *The AWK Programming Language*) as legitimate practice. With this exception, default argument lists have always been undocumented and vaguely defined, and it is not at all clear how (or if) they should be generalized to user-defined functions. They add no useful functionality and preclude possible future extensions that might need to name functions without calling them. Not standardizing them seems the simplest course. The standard developers considered that **length** merited special treatment, however, since it has been documented in the past and sees possibly substantial use in historical programs. Accordingly, this usage has been made legitimate, but Issue 5 removed the obsolescent marking for XSI-conforming implementations and many otherwise conforming applications depend on this feature.

In **sub** and **gsub**, if *repl* is a string literal (the lexical token **STRING**), then two consecutive backslash characters should be used in the string to ensure a single backslash will precede the ampersand when the resultant string is passed to the function. (For example, to specify one literal ampersand in the replacement string, use **gsub**(**ERE**, **"\\&"**).)

Historically the only special character in the *repl* argument of **sub** and **gsub** string functions was the ampersand (**'&'**) character and preceding it with the backslash character was used to turn off its special meaning.

The description in the ISO POSIX-2:1993 standard introduced behavior such that the backslash character was another special character and it was unspecified whether there were any other special characters. This description introduced several portability problems, some of which are described below, and so it has been replaced with the more historical description. Some of the problems include:

* Historically, to create the replacement string, a script could use **gsub**(**ERE**, **"\\&"**), but with the ISO POSIX-2:1993 standard wording, it was necessary to use **gsub**(**ERE**, **"\\\\&"**). Backslash characters are doubled here because all string literals are subject to lexical analysis, which would reduce each pair of backslash characters to a single backslash before being passed to **gsub**.

* Since it was unspecified what the special characters were, for portable scripts to guarantee that characters are printed literally, each character had to be preceded with a backslash. (For example, a portable script had to use **gsub**(**ERE**, **"\\h\\i"**) to produce a replacement string of **"hi"** .)

The description for comparisons in the ISO POSIX-2:1993 standard did not properly describe historical practice because of the way numeric strings are compared as numbers. The current rules cause the following code:

```
if (0 == "000")
  print "strange, but true"
else
  print "not true"
```

to do a numeric comparison, causing the **if** to succeed. It should be intuitively obvious that this is incorrect behavior, and indeed, no historical implementation of *awk* actually behaves this way.

To fix this problem, the definition of *numeric string* was enhanced to include only those values obtained from specific circumstances (mostly external sources) where it is not possible to determine unambiguously whether the value is intended to be a string or a numeric.

Variables that are assigned to a numeric string shall also be treated as a numeric string. (For example, the notion of a numeric string can be propagated across assignments.) In comparisons, all variables having the uninitialized value are to be treated as a numeric operand evaluating to the numeric value zero.

Uninitialized variables include all types of variables including scalars, array elements, and fields. The definition of an uninitialized value in Variables and Special Variables is necessary to describe the value placed on uninitialized variables and on fields that are valid (for example, **< $NF**) but have no characters in them and to describe how these variables are to be used in comparisons. A valid field, such as **$1**, that has no characters in it can be obtained from an input line of **"\t\t"** when **FS= '\t'** . Historically, the comparison (**$1<**10) was done numerically after evaluating **$1** to the value zero.

The phrase "... also shall have the numeric value of the numeric string" was removed from several sections of the ISO POSIX-2:1993 standard because is specifies an unnecessary implementation detail. It is not necessary for IEEE Std 1003.1-2001 to specify that these objects be assigned two different values. It is only necessary to specify that these objects may evaluate to two different values depending on context.

The description of numeric string processing is based on the behavior of the *atof*() function in the ISO C standard. While it is not a requirement for an implementation to use this function, many historical implementations of *awk* do. In the ISO C standard, floating-point constants use a period as a decimal point character for the language itself, independent of the current locale, but the *atof*() function and the associated *strtod*() function use the decimal point character of the current locale when converting strings to numeric values. Similarly in *awk*, floating-point constants in an *awk* script use a period independent of the locale, but input strings use the decimal point character of the locale.

FUTURE DIRECTIONS

None.

SEE ALSO

Grammar Conventions, *grep*, *lex*, *sed*, the System Interfaces volume of IEEE Std 1003.1-2001, *atof*(), *exec*, *popen*(), *setlocale*(), *strtod*()

COPYRIGHT