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 predefined 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. (Cur-
rent versions of the Bell Laboratories awk no longer accept them.)

-O
--optimize
Enable optimizations upon the internal representation of the program. Currently,
this includes just simple constant-folding. The gawk maintainer hopes to add addi-
tional optimizations over time.

-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 pro-
gram 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 particu-
larly 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 out-
put 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:

o \x escape sequences are not recognized.

o Only space and tab act as field separators when FS is set to a single space, new-
line does not.

o You cannot continue lines after ? and :.

o The synonym func for the keyword function is not recognized.

o The operators ** and **= cannot be used in place of ^ and ^=.

o 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 intermix-
ing 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 pro-
grams 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 dra-
conian 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-files 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 speci-
fied 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 assign-
ment. 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 IGNORE-CASE (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 mes-
sage.

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 redi-
rection 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 "groupn" for some n, which is the number of supplementary groups that the process has. Use the in operator to test for these elements. The follow-

ing 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 "FIELD-
WIDTHS" 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 char-
acter 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 sepa-
       rator 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 con-
       verted 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 con-
       stants.    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.

\xhex 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 reg-
       ular 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 char-
       acter 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 charac-
       ter 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 charac-
             ters, 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 repre-
      sented 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 "`"
      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," "',' and "`." In this case, [[=e=]] is a regular
expression that matches any of e, ', or `.

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 expres-
sions.

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 charac-
ter 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 1 on success, 0 on end of file, and -1
on an error.  Upon an
       error, ERRNO contains a string describing the problem.
```

NOTE: Failure in opening a two-way socket will result in a non-fatal error being returned to the calling function. 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 sup- pressed. 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.

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 posi-

tional 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 only to the 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 spa-
ces. 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 sys-
                            tem 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 dec-
imal, 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 get-
groups(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 <= 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 pro-
vided, 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 con-
tents of s are sorted using gawk's normal rules for comparing val-
ues, 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 charac-
ters in str translated to their corresponding lower-case counter-
parts. Non-alphabetic characters are left unchanged.

toupper(str) Returns a copy of the string str, with all the lower-case charac-
ters in str translated to their corresponding upper-case counter-
parts. 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 includ-
ing 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 mid-
night. 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 sys-time(). 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 cate-
gory. 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 sup-
ply 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_MES-
SAGES".
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 sup-
ply 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 for-
mal 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 every-
thing 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 ***--pro-***
file option. It then continues to run. SIGHUP causes pgawk to dump the profile and func-
tion 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 argu-
ments. 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. There-
fore 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 Labo-
ratories version); the tolower() and toupper() built-in functions (from the Bell Laborato-
ries 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 with-
out 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.

o No path search is performed for files named via the ***-f*** option. Therefore the AWKPATH environment variable is not special.

o The \x escape sequence. (Disabled with --posix.)

o The fflush() function. (Disabled with --posix.)

o The ability to continue lines after ? and :. (Disabled with --posix.)

o Octal and hexadecimal constants in AWK programs.

o The ARGIND, BINMODE, ERRNO, LINT, RT and TEXTDOMAIN variables are not special.

o The IGNORECASE variable and its side-effects are not available.

o The FIELDWIDTHS variable and fixed-width field splitting.

o The PROCINFO array is not available.

o The use of RS as a regular expression.

o The special file names available for I/O redirection are not recognized.

o The |& operator for creating co-processes.

o The ability to split out individual characters using the null string as the value of FS, and as the third argument to split().

o The optional second argument to the close() function.

o The optional third argument to the match() function.

o The ability to use positional specifiers with printf and sprintf().

o The ability to pass an array to length().

o The use of delete array to delete the entire contents of an array.

o The use of nextfile to abandon processing of the current input file.

o The and(), asort(), asorti(), bindtextdomain(), compl(), dcgettext(), dcngettext(), gen-

sub(), lshift(), mktime(), or(), rshift(), strftime(), strtonum(), systime() and xor() functions.

o Localizable strings.

o 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.

For socket communication, two special environment variables can be used to control the number of retries (GAWK_SOCK_RETRIES), and the interval between retries (GAWK_MSEC_SLEEP).

The interval is in milliseconds. On systems that do not support usleep(3), the value is rounded up to an integral number of seconds.

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.

EXIT STATUS

If the exit statement is used with a value, then gawk exits with the numeric value given to it.

Otherwise, if there were no problems during execution, gawk exits with the value of the C constant EXIT_SUCCESS. This is usually zero.

If an error occurs, gawk exits with the value of the C constant EXIT_FAILURE. This is usually one.

If gawk exits because of a fatal error, the exit status is 2. On non-POSIX systems, this value may be mapped to EXIT_FAILURE.

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.