NAME

nawk − pattern-directed scanning and processing language

SYNOPSIS

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

DESCRIPTION

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

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

A pattern-action statement has the form

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

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

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

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

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

The **print** statement prints its arguments on the standard output (or on a file if **>** *file* or **>>** *file* is present or on a pipe if **|***cmd* is present), separated by the current output field separator, and terminated by the output record separator. *file* and *cmd* may be literal names or parenthesized expressions; identical string values in different statements denote the same open file. The **printf** statement formats its expression list according to the format (see *printf* (3)). The built-in function **close(***expr***)** closes the file or pipe *expr*. The built-in function **fflush(***expr***)** flushes any buffered output for the file or pipe *expr*.

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

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

rand random number on (0,1)

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

int truncates to an integer value

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

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

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

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

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

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

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

system(*cmd*)
executes *cmd* and returns its exit status

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

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

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

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

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

A relational expression is one of the following:

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

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

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

Variable names with special meanings:

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

FS regular expression used to separate fields; also settable by option **–F** *fs.*

NF number of fields in the current record

NR ordinal number of the current record

FNR ordinal number of the current record in the current file

FILENAME
the name of the current input file

RS input record separator (default newline)

OFS output field separator (default blank)

ORS output record separator (default newline)

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

SUBSEP
separates multiple subscripts (default 034)

ARGC argument count, assignable

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

ENVIRON
array of environment variables; subscripts are names.

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

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

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

EXAMPLES

```
length($0) > 72
```
Print lines longer than 72 characters.

```
{ print $2, $1 }
```
Print first two fields in opposite order.

```
BEGIN { FS = ",[ \t]*|[ \t]+" }
      { print $2, $1 }
```
Same, with input fields separated by comma and/or blanks and tabs.

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

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

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

SEE ALSO

lex(1), *sed*(1)

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

BUGS

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

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	^=	**INCR**	++
OR	||	**DECR**	--
AND	&&	**APPEND**	>>

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

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

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: **?**, **:**, **,,** **^**
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