IEEE P1003.2 Draft 11.2 – September 1991

Copyright © 1991 by the
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change. The publication, distribution, or copying of this draft, as well as all derivative works based on this draft, is expressly prohibited except as set forth below.

Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities only, and subject to the restrictions contained herein.

Permission is hereby also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position, subject to the restrictions contained herein.

Permission is hereby also granted to the preceding entities to make limited copies of this document in an electronic form only for the stated activities.

The following restrictions apply to reproducing or transmitting the document in any form: 1) all copies or portions thereof must identify the document's IEEE project number and draft number, and must be accompanied by this entire notice in a prominent location; 2) no portion of this document may be redistributed in any modified or abridged form without the prior approval of the IEEE Standards Department.

Other entities seeking permission to reproduce this document, or any portion thereof, for standardization or other activities, must contact the IEEE Standards Department for the appropriate license.

Use of information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

Section 4: Execution Environment Utilities

The Execution Environment Utilities are the utilities that shall be implemented in all conforming POSIX. 2 systems.

4.1 awk — Pattern scanning and processing language

4.1.1 Synopsis

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

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

4.1.2 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 shall be carried out.

Input shall be interpreted as a sequence of records. By default, a record is a line, 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>` characters. This default white space field delimiter can be changed by using the `FS` built-in variable or the `-F` *ERE* . The `awk` utility shall denote the first field in a record `$1`, the second `$2`, and so forth. The symbol `$0` shall refer to the entire record; setting any other field shall cause the reevaluation of `$0`. Assigning to `$0` shall reset the values of all other fields and the `NF` built-in variable.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.1.3 Options

The awk utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−F *ERE* Define the input field separator to be the extended regular expression *ERE* , before any input is read (see 4.1.7.4).

−f *progfile* Specifies 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 *assignment* argument shall be 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.

4.1.4 Operands

The following operands shall be supported by the implementation:

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> character, awk shall interpret the text as if it did.

argument Either of the following two types of *argument* s 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 Table 2-3 in 2.4), 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 = shall represent the name of an awk variable; if that name is an awk reserved word (see 4.1.7.7) the behavior is undefined. The characters following the equals-sign shall be interpreted as if they appeared in the awk program preceded and followed by a double-

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

quote (") character, as a STRING token (see 4.1.7.7), 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. If that value is considered a *numeric string* (see 4.1.7.2), 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.

4.1.5 External Influences

4.1.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is –. See Input Files.

4.1.5.2 Input Files

Input files to the awk program from any of the following sources:

— 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

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

If –f *progfile* is specified, the file(s) named by *progfile* shall be text file(s) containing an awk program.

4.1.5.3 Environment Variables

The following environment variables shall affect the execution of awk:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files), the behavior of character classes within regular expressions, the identification of characters as letters, and the mapping of upper- and lowercase characters for the `toupper` and `tolower` functions.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions and in comparisons of string values.

LC_MESSAGES This variable shall determine the language in which messages should be written.

LC_NUMERIC This variable shall determine the radix character used when interpreting numeric input, performing conversions between numeric and string values, and formatting numeric output.

PATH This variable shall define the search path when looking for commands executed by `system(`*expr*`)`, or input and output pipes. See 2.6.

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

4.1.5.4 Asynchronous Events

Default.

4.1.6 External Effects

4.1.6.1 Standard Output

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

4.1.6.2 Standard Error

Used only for diagnostic messages.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.1.6.3 Output Files

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

4.1.7 Extended Description

4.1.7.1 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 an action that writes the matched record of input to standard output.

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), splitting the current record into fields using the current value of FS according to the rules in 4.1.7.4, evaluating each pattern in the program in the order of occurrence, and executing the action associated with each pattern that matches the current record. The action for a matching pattern shall be executed before evaluating subsequent patterns. Last, the actions associated with all END patterns shall be executed in the order they occur in the program.

4.1.7.2 Expressions

Expressions describe computations used in *patterns* and *actions.* In Table 4-1, 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, 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 (i.e., on the left side of an assignment operator). The precise syntax of expressions is given in the grammar in 4.1.7.7.

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 C Standard {7}:

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

A numeric value that is exactly equal to the value of an integer (see 2.9.2.1) shall

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 4-1 – awk Expressions in Decreasing Precedence

Syntax	Name	Semantic Definition	Type of Result	Assoc
(*expr*)	Grouping	C Standard {7}	type of *expr*	n/a
$*expr*	Field reference	4.1.7.2	string	n/a
++ *lvalue*	Pre-increment	C Standard {7}	numeric	n/a
−− *lvalue*	Pre-decrement	C Standard {7}	numeric	n/a
lvalue ++	Post-increment	C Standard {7}	numeric	n/a
lvalue −−	Post-decrement	C Standard {7}	numeric	n/a
expr ^ *expr*	Exponentiation	4.1.7.2	numeric	right
! *expr*	Logical not	C Standard {7}	numeric	n/a
+ *expr*	Unary plus	C Standard {7}	numeric	n/a
− *expr*	Unary minus	C Standard {7}	numeric	n/a
expr * *expr*	Multiplication	C Standard {7}	numeric	left
expr / *expr*	Division	C Standard {7}	numeric	left
expr % *expr*	Modulus	4.1.7.2	numeric	left
expr + *expr*	Addition	C Standard {7}	numeric	left
expr − *expr*	Subtraction	C Standard {7}	numeric	left
expr expr	String concatenation	4.1.7.2	string	left
expr < *expr*	Less than	4.1.7.2	numeric	none
expr <= *expr*	Less than or equal to	4.1.7.2	numeric	none
expr != *expr*	Not equal to	4.1.7.2	numeric	none
expr == *expr*	Equal to	4.1.7.2	numeric	none
expr > *expr*	Greater than	4.1.7.2	numeric	none
expr >= *expr*	Greater than or equal to	4.1.7.2	numeric	none
expr ~ *expr*	ERE match	4.1.7.4	numeric	none
expr !~ *expr*	ERE nonmatch	4.1.7.4	numeric	none
expr in array	Array membership	4.1.7.2	numeric	left
(*index*) in *array*	Multidimension array membership	4.1.7.2	numeric	left
expr && *expr*	Logical AND	C Standard {7}	numeric	left
expr || *expr*	Logical OR	C Standard {7}	numeric	left
expr1 ? *expr2* : *expr3*	Conditional expression	C Standard {7}	type of selected *expr2* or *expr3*	right
lvalue ^= *expr*	Exponentiation assignment	4.1.7.2	numeric	right
lvalue %= *expr*	Modulus assignment	4.1.7.2	numeric	right
lvalue *= *expr*	Multiplication assignment	C Standard {7}	numeric	right
lvalue /= *expr*	Division assignment	C Standard {7}	numeric	right
lvalue += *expr*	Addition assignment	C Standard {7}	numeric	right
lvalue −= *expr*	Subtraction assignment	C Standard {7}	numeric	right
lvalue = *expr*	Assignment	C Standard {7}	type of *expr*	right

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

be converted to a string by the equivalent of a call to the `sprintf` function (see 4.1.7.6.2) 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 standard 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 to be a *numeric string* in the following case:

(1) Any leading and trailing `<blank>`s shall be ignored.

(2) If the first unignored character is a `+` or –, it shall be ignored.

(3) If the remaining unignored characters would be lexically recognized as a `NUMBER` token (as described by the lexical conventions in 4.1.7.7), the string shall be considered a *numeric string* .

If a – character is ignored in the above steps, 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 clause.

When an expression is used in a Boolean context (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, or the expression of a `while` statement), 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.

All arithmetic shall follow the semantics of floating point arithmetic as specified by the C Standard {7};see 2.9.2.

The value of the expression

expr1 `^` *expr2*

shall be equivalent to the value returned by the C Standard {7}function call

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

The expression

lvalue `^=` *expr*

shall be equivalent to the C Standard {7}expression

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

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

255 *expr1* % *expr2*

256 shall be equivalent to the value returned by the C Standard {7} function call

257 fmod(*expr1* , *expr2*)

258 The expression

259 *lvalue* %= *expr*

260 shall be equivalent to the C Standard {7} expression

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

262 except that *lvalue* shall be evaluated only once.

263 Variables and fields shall be set by the assignment statement:

264 *lvalue* = *expression*

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

270 The awk language shall supply arrays that are used for storing numbers or
271 strings. Arrays need not be declared. They shall initially be empty, and their
272 sizes shall change dynamically. The subscripts, or element identifiers, are
273 strings, providing a type of associative array capability. An array name followed
274 by a subscript within square brackets can be used as an *lvalue* and thus as an
275 expression, as described in the grammar (see 4.1.7.7). Unsubscripted array
276 names can be used in only the following contexts:

277 — A parameter in a function definition or function call.

278 — The NAME token following any use of the keyword in as specified in the
279 grammar (see 4.1.7.7). If the name used in this context is not an array
280 name, the behavior is undefined.

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

288 *var* [*expr1* , *expr2* , ... , *exprn*]
289 *var* [*expr1* SUBSEP *expr2* SUBSEP ... SUBSEP *exprn*]

290 A multidimensioned *index* used with the in operator shall be parenthesized. The
291 in operator, which tests for the existence of a particular array element, shall not
292 cause that element to exist. Any other reference to a nonexistent array element
293 shall automatically create it.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Comparisons (with the <, <=, !=, ==, >, and >= operators) shall be made numerically if both operands are numeric or if one is numeric and the other has a string value that is a numeric string. 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.

4.1.7.3 Variables and Special Variables

Variables can be used in an awk program by referencing them. With the exception of function parameters (see 4.1.7.6.2), they are not explicitly declared. Uninitialized scalar variables and array elements have both a numeric value of zero and a string value of the empty string.

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 nonnegative 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 (i.e., fields after $NF), shall produce the null string. However, assigning to a nonexistent field [e.g., $(NF+2) = 5] shall increase the value of NF, create any intervening fields with the null string as their values, 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 when created. If the string, with any occurrence of the decimal-point character from the current locale changed to a <period>, would be considered a *numeric string* (see 4.1.7.2), the field variable shall also have the numeric value of the *numeric string* .

The implementation 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 nonnull element of ARGV, up through the current value of ARGC−1, 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 '-' shall indicate 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 — The variable ENVIRON is an array representing the value of the environment, as described in POSIX .1 {8} 2.7. 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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

consisting of the value of that variable. If the value of an environment variable is considered a *numeric string* (see 4.1.7.2), 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 command(s) 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 is 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 is zero. Inside an END action the value is the number of the last record processed in the last file processed.

FS Input field separator regular expression; <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 is zero. Inside an END action the value is the number of the last record processed.

OFMT The printf format for converting numbers to strings in output statements (see 4.1.7.6.1); "%.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; <newline> by default.

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

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

380 RSTART The starting position of the string matched by the match func-
381 tion, numbering from 1. This is always equivalent to the return
382 value of the match function.

383 SUBSEP The subscript separator string for multidimensional arrays; the
384 default value is implementation defined.

385 #### 4.1.7.4 Regular Expressions

386 The awk utility shall make use of the extended regular expression notation (see
387 2.8.4) except that it shall allow the use of C-language conventions for escaping
388 special characters within the ERE s, as specified in Table 2-15 and Table 4-2; these 1
389 escape sequences shall be recognized both inside and outside bracket expressions. 1
390 Note that records need not be separated by <newline>s and string constants can 1
391 contain <newline>s, so even the \n sequence is valid in awk ERE s. Using a slash 1
392 character within the regular expression requires the escaping shown in Table 4-2. 1

393 A regular expression can be matched against a specific field or string by using one
394 of the two regular expression matching operators, ~ and !~. These operators shall
395 interpret their right-hand operand as a regular expression and their left-hand
396 operand as a string. If the regular expression matches the string, the ~ expres-
397 sion shall evaluate to a value of 1, and the !~ expression shall evaluate to a value
398 of 0. (The regular expression matching operation is as defined in 2.8.1.2, where a
399 match occurs on any part of the string unless the regular expression is limited
400 with the circumflex or dollar-sign special characters.) If the regular expression
401 does not match the string, the ~ expression shall evaluate to a value of 0, and the
402 !~ expression shall evaluate to a value of 1. If the right-hand operand is any
403 expression other than the lexical token ERE, the string value of the expression
404 shall be interpreted as an extended regular expression, including the escape con-
405 ventions described above. Note that these same escape conventions also shall be
406 applied in the determining the value of a string literal (the lexical token STRING),
407 and thus shall be applied a second time when a string literal is used in this con-
408 text.

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

412 `$0 ~ /`*ere*`/`

413 The *ERE* argument to the gsub, match, sub functions, and the *fs* argument to the
414 split function (see 4.1.7.6.2) shall be interpreted as extended regular expres-
415 sions. These can be either ERE tokens or arbitrary expressions, and shall be inter-
416 preted in the same manner as the right-hand side of the ~ or !~ operator.

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

(1) 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*.

(2) 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 in the gsub, match, split, and sub built-in functions, regular expression matching shall be based on input records; i.e., 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> characters embedded in the expression can be matched. In those four built-in functions, regular expression matching shall be based on text strings; i.e., 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 regular expression matching, the use of one or more NUL characters in the pattern, input record, or text string produces undefined results.

4.1.7.5 Patterns

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

4.1.7.5.1 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 4.1.7.6.2) 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 action(s) are executed.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.1.7.5.2 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.

4.1.7.5.3 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.

4.1.7.6 Actions

An action is a sequence of statements as shown in the grammar in 4.1.7.7. Any single statement can be replaced by a statement list enclosed in braces. The statements in a statement list shall be separated by <newline>s or semicolons, and 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 nonzero or nonnull, 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 C Standard {7} (see 2.9.2), except that the Boolean expressions shall be treated as described in 4.1.7.2, 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 shall delete 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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

executed.

4.1.7.6.1 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 full pathname to write into (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 simply shall append output to the file. The file remains open until the `close` function (see 4.1.7.6.2). 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 (see B.3.2) 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 4.1.7.6.2) 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 (see B.3.2).

As described in detail by the grammar in 4.1.7.7, these output statements shall take a comma-separated list of *expression* s referred in the grammar by the nonterminal 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 4.1.7.2, 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 standard (see 2.12). Output shall be produced as specified with the first expression argument as the string *<format>* and subsequent expression arguments as the strings *<arg1>* through *<argn>*, with the following exceptions:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

(1) The *format* shall be an actual character string rather than a graphical representation. Therefore, it cannot contain empty character positions. The <space> character 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 Δ 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 specifications with <blank>s not specified by the *format* string.

(6) The implementation shall not precede output from the o conversion specification with leading zeroes not specified by the *format* string.

(7) For the c conversion specification: 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, the value shall be converted (according to the rules specified in 4.1.7.2) 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.

4.1.7.6.2 Functions

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.1.7.6.2.1 Arithmetic Functions

The arithmetic functions, except for `int`, shall be based on the C Standard {7};see 2.9.2. The behavior is undefined in cases where the C Standard {7} specifies that an error be returned or that the behavior is undefined.

`atan2`(*y*,*x*)	Return arctangent of y/x.
`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*)	Truncate its argument to an integer. It shall be truncated toward 0 when $x > 0$.
`rand()`	Return a random number n, such that $0 \le n < 1$.
`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.

4.1.7.6.2.2 String Functions

The string functions are:

`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 $a[1], a[2], \ldots, a[n]$, and returns n. The separation shall be done with the extended regular expression *fs* or with the field separator

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

FS if *fs* is not given. Each array element shall have a string value when created. If the string assigned to any array element, with any occurrence of the decimal-point character from the current locale changed to a <period>, would be considered a *numeric string* (see 4.1.7.2), the array element shall also have the numeric value of the *numeric string* . 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 regular expression. An ampersand preceded by a backslash within *repl* shall be interpreted as a literal ampersand character. If *in* is specified and it is not an *lvalue* (see 4.1.7.2), the behavior is undefined. If *in* is omitted, awk shall substitute in the current record ($0).

substr(*s*, *m*[, *n*])
Return the at most *n* -character substring of *s* that begins at position *m* , numbering from 1. If *n* is missing, 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 shall be replaced in the returned string by the uppercase letter specified by the mapping. Other characters in *s* shall be 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 4.1.7.4.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.1.7.6.2.3 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 nonzero.

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 file. 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 missing, `$0` and `NF` shall be set; otherwise, *var* shall be set.

`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. 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 full 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 missing, `$0` and `NF` shall be set; otherwise, *var* shall be set.

`system(`*expression* `)` Execute the command given by *expression* in a manner equivalent to the *system* () function [see B.3.1] and return

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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.

4.1.7.6.2.4 User-Defined Functions

The `awk` language also shall provide user-defined functions. Such functions can be defined as:

`function` *name* `(`*args*`,...) {` *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 shall be global.

Function arguments can be either scalars or arrays; the behavior is undefined if an array name is passed as an argument that the function uses as a scalar, or if a scalar expression is passed as an argument that the function uses as an array. Function arguments shall be passed by value if scalar and by reference if array name. Argument names shall be local to the function; all other variable names shall be global. The same name shall not be used as both an argument 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.

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 be initialized with a string value of the null string and a numeric value of zero, and the extra parameters that are used in the function body as arrays shall be initialized as empty arrays. If more arguments are supplied in a function call than are in the function definition, the behavior is undefined.

When invoking a function, no white space can be placed between the function name and the opening parenthesis. The implementation shall permit function calls to be nested, and for recursive calls to 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.

4.1.7.7 `awk` Grammar

The grammar in this subclause and the lexical conventions in the following subclause shall together describe the syntax for `awk` programs. The general conventions for this style of grammar are described in 2.1.2. A valid program can be

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

represented as the nonterminal symbol *program* in the grammar. Any discrepancies found between this grammar and other descriptions in this clause shall be resolved in favor of this grammar.

```
%token  NAME NUMBER STRING ERE NEWLINE
%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  '{' '}' '(' ')' '[' ']' ',' ';'
%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
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
796         | Function NAME '(' param_list_opt ')' newline_opt action
797         | Function FUNC_NAME '(' param_list_opt ')' newline_opt action
798         ;
799     param_list_opt:
800           /* empty */
801         | param_list
802         ;
803     param_list:
804           NAME
805         | param_list ',' NAME
806         ;
807     pattern:
808           Begin
809         | End
810         | expr
811         | expr ',' newline_opt expr
812         ;
813     action:
814           '{' newline_opt '}'
815         | '{' newline_opt terminated_statement_list '}'
816         | '{' newline_opt unterminated_statement_list '}'
817         ;
818     terminator:
819           ';'
820         | NEWLINE
821         | terminator NEWLINE ';'                                                    2
822         ;
823     terminated_statement_list:
824           terminated_statement
825         | terminated_statement_list terminated_statement
826         ;
827     unterminated_statement_list:
828           unterminated_statement
829         | terminated_statement_list unterminated_statement
830         ;
831     terminated_statement:
832           action newline_opt
833         | If '(' expr ')' newline_opt terminated_statement
834           Else newline_opt terminated_statement
835         | While '(' expr ')' newline_opt terminated_statement
836         | For '(' simple_statement_opt ';' expr_opt ';' simple_statement_opt ')'
837           newline_opt terminated_statement
838         | For '(' NAME In NAME ')' newline_opt terminated_statement
839         | ';' newline_opt
840         | terminatable_statement NEWLINE newline_opt
841         | terminatable_statement ';' newline_opt
842         ;
843     unterminated_statement:
844           terminatable_statement
845         | If '(' expr ')' newline_opt unterminated_statement
846         | If '(' expr ')' newline_opt terminated_statement
847           Else newline_opt unterminated_statement
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
848         | While '(' expr ')' newline_opt unterminated_statement
849         | For '(' simple_statement_opt ';' expr_opt ';' simple_statement_opt ')'
850           newline_opt unterminated_statement
851         | For '(' NAME In NAME ')' newline_opt unterminated_statement
852         ;
853     terminatable_statement:
854           simple_statement
855         | Break
856         | Continue
857         | Next
858         | Exit expr_opt
859         | Return expr_opt
860         | Do newline_opt terminated_statement While '(' expr ')'
861         ;
862     simple_statement_opt:
863           /* empty */
864         | simple_statement
865         ;
866     simple_statement:
867           Delete NAME '[' expr_list ']'
868         | expr
869         | print_statement
870         ;
871     print_statement:
872           simple_print_statement
873         | simple_print_statement output_redirection
874         ;
875     simple_print_statement:
876           Print print_expr_list_opt
877         | Print '(' multiple_expr_list ')'
878         | Printf print_expr_list
879         | Printf '(' multiple_expr_list ')'
880         ;
881     output_redirection:
882           '>' expr
883         | APPEND expr
884         | '|' expr
885         ;
886     expr_list_opt:
887           /* empty */
888         | expr_list
889         ;
890     expr_list:
891           expr
892         | multiple_expr_list
893         ;
894     multiple_expr_list:
895           expr ',' newline_opt expr
896         | multiple_expr_list ',' newline_opt expr
897         ;
898     expr_opt:
899           /* empty */
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
900         | expr
901         ;
902     expr:
903           unary_expr
904         | non_unary_expr
905         ;
906     unary_expr:
907           '+' expr
908         | '-' expr
909         | unary_expr '^' expr
910         | unary_expr '*' expr
911         | unary_expr '/' expr
912         | unary_expr '%' expr
913         | unary_expr '+' expr
914         | unary_expr '-' expr
915         | unary_expr non_unary_expr
916         | unary_expr '<' expr
917         | unary_expr LE expr
918         | unary_expr NE expr
919         | unary_expr EQ expr
920         | unary_expr '>' expr
921         | unary_expr GE expr
922         | unary_expr '~' expr
923         | unary_expr NO_MATCH expr
924         | unary_expr In NAME
925         | unary_expr AND newline_opt expr
926         | unary_expr OR newline_opt expr
927         | unary_expr '?' expr ':' expr
928         | unary_input_function
929         ;
930     non_unary_expr:
931           '(' expr ')'
932         | '!' expr
933         | non_unary_expr '^' expr
934         | non_unary_expr '*' expr
935         | non_unary_expr '/' expr
936         | non_unary_expr '%' expr
937         | non_unary_expr '+' expr
938         | non_unary_expr '-' expr
939         | non_unary_expr non_unary_expr
940         | non_unary_expr '<' expr
941         | non_unary_expr LE expr
942         | non_unary_expr NE expr
943         | non_unary_expr EQ expr
944         | non_unary_expr '>' expr
945         | non_unary_expr GE expr
946         | non_unary_expr '~' expr
947         | non_unary_expr NO_MATCH expr
948         | non_unary_expr In NAME
949         | '(' multiple_expr_list ')' In NAME
950         | non_unary_expr AND newline_opt expr
951         | non_unary_expr OR newline_opt expr
952         | non_unary_expr '?' expr ':' expr
953         | NUMBER
954         | STRING
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
955         | lvalue
956         | ERE
957         | lvalue INCR
958         | lvalue DECR
959         | INCR lvalue
960         | DECR lvalue
961         | lvalue POW_ASSIGN expr
962         | lvalue MOD_ASSIGN expr
963         | lvalue MUL_ASSIGN expr
964         | lvalue DIV_ASSIGN expr
965         | lvalue ADD_ASSIGN expr
966         | lvalue SUB_ASSIGN expr
967         | lvalue '=' expr
968         | FUNC_NAME '(' expr_list_opt ')'  /* no white space allowed */
969         | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
970         | BUILTIN_FUNC_NAME
971         | non_unary_input_function
972         ;
973     print_expr_list_opt:
974           /* empty */
975         | print_expr_list
976         ;
977     print_expr_list:
978           print_expr
979         | print_expr_list ',' newline_opt print_expr
980         ;
981     print_expr:
982           unary_print_expr
983         | non_unary_print_expr
984         ;
985     unary_print_expr:
986           '+' print_expr
987         | '-' print_expr
988         | unary_print_expr '^' print_expr
989         | unary_print_expr '*' print_expr
990         | unary_print_expr '/' print_expr
991         | unary_print_expr '%' print_expr
992         | unary_print_expr '+' print_expr
993         | unary_print_expr '-' print_expr
994         | unary_print_expr non_unary_print_expr
995         | unary_print_expr '~' print_expr
996         | unary_print_expr NO_MATCH print_expr
997         | unary_print_expr In NAME
998         | unary_print_expr AND newline_opt print_expr
999         | unary_print_expr OR newline_opt print_expr
1000        | unary_print_expr '?' print_expr ':' print_expr
1001        ;
1002    non_unary_print_expr:
1003          '(' expr ')'
1004        | '!' print_expr
1005        | non_unary_print_expr '^' print_expr
1006        | non_unary_print_expr '*' print_expr
1007        | non_unary_print_expr '/' print_expr
1008        | non_unary_print_expr '%' print_expr
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
      | 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 */
      | 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:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

— Operator precedence and associativity shall be as described in Table 4-1.

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

4.1.7.8 `awk` 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>` character. Comments shall have no effect, except to delimit lexical tokens.

(3) The character `<newline>` shall be recognized as the token `NEWLINE`.

(4) A backslash character immediately followed by a `<newline>` character 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 Table 2-15 (see 2.12). In addition, the escape sequences in Table 4-2 shall be recognized. A `<newline>` character 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 Table 2-15 (see 2.12). In addition, the escape sequences in Table 4-2 shall be recognized. A `<newline>` character shall 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 string 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 the either of the tokens `floating-constant` or `integer-constant` as specified by the C Standard {7}, with the following exceptions:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 4-2 – awk Escape Sequences

Escape Sequence	Description	Meaning
\"	<backslash> <quotation-mark>	<quotation-mark> character
\/	<backslash> <slash>	<slash> character
ddd	<backslash> followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0, (i.e., representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the one-, two-, or three-digit octal integer. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation defined. Multibyte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.
c	<backslash> followed by any character not described in this table or in Table 2-15	Undefined

(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 2.9.2.1), the behavior is undefined.

(9) A sequence of underscores, digits, and alphabetics from the portable character set (see 2.4), 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   for      in     printf
END       do       function next   return
break     else     getline  print  while
continue  exit     if
```

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

```
atan2   index   match   sprintf   substr
close   int     rand    sqrt      system
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
cos     length  sin     srand     tolower
exp     log     split   sub       toupper
gsub
```

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.

4.1.8 Exit Status

The awk utility shall exit with one of the following values:

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.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.1.9 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 the *progfile* operand(s) is not a valid `awk` program (as specified in 4.1.7), the behavior is undefined.

4.1.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `awk` program specified in the command line is most easily specified within single-quotes (e.g., *'program '*) for applications using `sh`, because `awk` 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) Write any line in which the second field matches the regular expression and the fourth field does not.

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

(5) Write any line in which the second field contains a backslash.

```
$2 ~ /\\/
```

(6) 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 ~ "\\\\"
```

(7) Write the second to the last and the last field in each line. Separate the fields by a colon.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

(8) 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}
```

(9) Write lines longer than 72 characters.

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

(10) Write first two fields in opposite order separated by the `OFS`:

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

(11) Same, with input fields separated by comma and/or `<space>`s and `<tab>`s:

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

(12) Add up first column, print sum and average.

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

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

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

(14) Write all lines between occurrences of the strings `start` and `stop`:

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

(15) Write all lines whose first field is different from the previous one:

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

(16) Simulate `echo`:

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

(17) 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]
}
```

(18) If there is a file named ‘‘input’’ containing page headers of the form:

```
Page #
```

and a file named ‘‘program’’ that contains:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

then the command line:

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

will print the file ''input,'' filling in page numbers starting at 5.

The index, length, match, and substr should not be confused with similar functions in the C Standard {7}; the awk versions deal with characters, while the C Standard {7} deals with bytes.

To forestall any possible confusion, where strings are used as the name of a file or pipeline, the strings must be textually identical. The terminology ''same string value'' implies that ''equivalent strings,'' even those that differ only by <space>s, represent different files.

History of Decisions Made

This description is based on the new awk, 'nawk,'' (see *The AWK Programming Language* {B21}), which introduced a number of new features to the historical awk:

(1) New keywords: delete, do, function, return

(2) New built-in functions: atan2, cos, sin, rand, srand, gsub, sub, match, close, 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 are now treated as extended regular expressions.

(6) The operator precedence has changed to more closely match C. 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:

(1) Multiple instances of -f *progfile* are permitted.

(2) New option: -v *assignment*

(3) New predefined variable: ENVIRON

(4) New built-in functions: toupper, tolower

(5) More formatting capabilities added to printf to match the C Standard {7}.

Regular expressions have been extended somewhat from traditional implementations to make them a pure superset of Extended Regular Expressions as defined by this standard (see 2.8.4). The main extensions are internationalization

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

features and interval expressions. Traditional implementations of awk have long supported <backslash> escape sequences as an extension to regular expressions, and this extension has been retained despite inconsistency with other utilities. The number of escape sequences recognized in both regular expressions and strings has varied (generally increasing with time) among implementations. The set specified by the standard includes most sequences known to be supported by popular implementations and by the C Standard {7}. 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 C Standard {7}. However, because this syntax has a nondeterministic length, it does not permit the subsequent character to be a hexadecimal digit. This limitation can be worked around 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 regular expressions (either lexical ERE tokens or strings used dynamically as regular expressions). Because of this limitation, the feature has not been added to POSIX. 2.

When a string variable is used in a context where an ERE 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 language's authors, who have asked that it not be included in the standard.

Traditional 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 traditional implementations or in *The AWK Programming Language* {B21}. 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 existing practice in almost all cases. The one exception is that, in traditional 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 traditional implementations the following program:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
{
        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. POSIX.2 ensures that comparisons will be numeric if necessary. With traditional 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. The standard requires that the output of the second `print` statement be **3.140000**. The behavior of traditional implementations was seen as too unintuitive and unpredictable.

However, a further modification was made in Draft 11. It was pointed out that with the Draft 10 rules, 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 existing 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, logical AND operator

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

(&&), 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.

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.

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 this standard to address such issues. The standard has made a few relatively minor changes toward making the language more compatible with the C language as specified by the C Standard {7}; 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.

This standard requires several changes from traditional 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 C Standard {7}. 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 POSIX. 2. It is possible to explicitly execute an `awk` program in any desired locale by setting the environment in the shell.

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 nonzero exit status. However, some implementations may choose to extend the language in ways that make use of certain invalid constructs.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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 multicharacter `RS` values is to allow possible future extensions based on regular expressions used for record separators. Historical implementations take the first character of the string and ignore the others.

The undefined behavior resulting from NUL s in regular expressions allows future extensions for the GNU `gawk` program to process binary data.

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.

4.2 `basename` — Return nondirectory portion of pathname

4.2.1 Synopsis

`basename` *string* [*suffix*]

4.2.2 Description

The *string* operand shall be treated as a pathname, as defined in 2.2.2.102. The string *string* shall be converted to the filename corresponding to the last pathname component in *string* and then the suffix string *suffix* , if present, shall be removed. This shall be done by performing actions equivalent to the following steps in order:

(1) If *string* is `//`, it is implementation defined whether steps (2) through (5) are skipped or processed.

(2) If *string* consists entirely of slash characters, *string* shall be set to a single slash character. In this case, skip steps (3) through (5).

(3) If there are any trailing slash characters in *string* , they shall be removed.

(4) If there are any slash characters remaining in *string* , the prefix of *string* up to and including the last slash character in *string* shall be removed.

(5) If the *suffix* operand is present, is not identical to the characters remaining in *string* , and is identical to a suffix of the characters remaining in *string* , the suffix *suffix* shall be removed from *string* . Otherwise, *string*

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.