IEEE P1003.2 Draft 11.2 − September 1991

The `unset` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

0 All *name* s were successfully unset.

>0 At least one *name* could not be unset.

3.14.14.1 `unset` Rationale. *(This subclause is not a part of P1003.2)*

Note that

```
VARIABLE=
```

is not equivalent to an `unset` of `VARIABLE`; in the example, `VARIABLE` is set to `""`. Also, the ‘‘variables’’ that can be `unset` should not be misinterpreted to include the special parameters (see 3.5.2).

Consideration was given to omitting the `−f` option in favor of an `unfunction` utility, but decided to retain existing practice.

The `−v` option was introduced because System V historically used one name space 1
for both variables and functions. When `unset` is used without options, System V 1
historically unset either a function or a variable and there was no confusion about 1
which one was intended. A portable POSIX.2 application can use `unset` without 1
an option to unset a variable, but not a function; the `−f` option must be used. 1

Examples :

Unset the **VISUAL** variable:

```
unset -v VISUAL
```
1

Unset the functions `foo` and `bar`:

```
unset −f foo bar
```

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.

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-

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.

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.

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

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	1
expr || *expr*	Logical OR	C Standard {7}	numeric	left	1
expr1 ? *expr2* : *expr3*	Conditional expression	C Standard {7}	type of selected *expr2* or *expr3*	right	1
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	

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 1
unspecified if the value of `CONVFMT` is not a floating-point format specification. 1
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 *expr1* `%` *expr2*

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

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

The expression

lvalue `%=` *expr*

shall be equivalent to the C Standard {7} 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 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 shall supply 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 4.1.7.7). 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 4.1.7.7). 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 multidimensional 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* `]`

A multidimensioned *index* used with the `in` operator shall be 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 or if one is numeric and the other has a string value that is a numeric string. Otherwise, operands shall be converted to strings 1 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 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.

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

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

4.1.7.4 Regular Expressions

The `awk` utility shall make use of the extended regular expression notation (see 2.8.4) except that it shall allow the use of C-language conventions for escaping special characters within the ERE s, as specified in Table 2-15 and Table 4-2; 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` ERE s. Using a slash character within the regular expression requires the escaping shown in Table 4-2.

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 in 2.8.1.2, 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 also shall be applied in the 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 4.1.7.6.2) 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>` character. The following describes `FS` behavior:

(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.

4.1.7.5.2 Expression Patterns

An expression pattern shall be evaluated as if it were an expression in a Boolean 1
context. If the result is true, the pattern shall be considered to match, and the 1
associated action (if any) shall be executed. If the result is false, the action shall 1
not be executed. 1

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 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 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 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:

(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.

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], . . . , *a*[*n*], and returns *n*. The separation shall be done with the extended regular expression *fs* or with the field separator

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.

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