NAME

nawk – pattern scanning and processing language

SYNOPSIS

/usr/bin/nawk [**–F** *ERE*] [**–v** *assignment*] *'program'* | **–f** *progfile* . . . [*argument* . . .]

/usr/xpg4/bin/awk [**–F** *ERE*] [**–v** *assignment* . . .] *'program'* | **–f** *progfile* . . .
[*argument* . . .]

AVAILABILITY

SUNWesu

/usr/xpg4/bin/awk

SUNWxcu4

DESCRIPTION

The **/usr/bin/nawk** and **/usr/xpg4/bin/awk** utilities execute *program*s written in the *nawk* programming language, which is specialized for textual data manipulation. A **nawk** *program* is a sequence of patterns and corresponding actions. The string specifying *program* must be enclosed in single quotes (′) to protect it from interpretation by the shell. The sequence of pattern - action statements can be specified in the command line as *program* or in one, or more, file(s) specified by the **-f** *progfile* option. When input is read that matches a pattern, the action associated with the pattern is performed.

Input is 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 is matched to each pattern in the *program*. For each pattern matched, the associated action is executed.

The **nawk** utility interprets 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 (blanks and/or tabs) can be changed by using the **FS** built-in variable or the –**F** *ERE* option. The **nawk** utility denotes the first field in a record **$1**, the second **$2**, and so forth. The symbol **$0** refers to the entire record; setting any other field causes the reevaluation of **$0**. Assigning to **$0** resets the values of all fields and the **NF** built-in variable.

OPTIONS

The following options are supported:

–**F** *ERE* Define the input field separator to be the extended regular expression *ERE*, before any input is read (can be a character).

–**f** *progfile* Specifies the pathname of the file *progfile* containing a **nawk** program. If multiple instances of this option are specified, the concatenation of the files specified as *progfile* in the order specified is the **nawk** program. The **nawk** program can alternatively be specified in the command line as a single argument.

–**v** *assignment* The *assignment* argument must be in the same form as an *assignment* operand. The assignment is of the form *var=value*, where *var* is the name of one of the variables described below. The specified assignment occurs before executing the **nawk** program, including the actions associated with **BEGIN** patterns (if any). Multiple occurrences of this option can be specified.

OPERANDS

The following operands are supported:

program If no **–f** option is specified, the first operand to **nawk** is the text of the **nawk** program. The application supplies the *program* operand as a single argument to **nawk.** If the text does not end in a newline character, **nawk** interprets the text as if it did.

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

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

assignment An operand that begins with an underscore or alphabetic character from the portable character set, followed by a sequence of underscores, digits and alphabetics from the portable character set, followed by the **=** character specifies a variable assignment rather than a pathname. The characters before the **=** represent the name of a **nawk** variable; if that name is a **nawk** reserved word the behavior is undefined. The characters following the equal sign is interpreted as if they appeared in the **nawk** program preceded and followed by a double-quote (**"**) character, as a **STRING** token , except that if the last character is an unescaped backslash, it is interpreted as a literal backslash rather than as the first character of the sequence " .}S 1 3 "**" " ".'''' '''' '''' ''''** **''''** The variable is assigned the value of that **STRING** token. If the value is considered a *numeric*string, the variable is assigned its numeric value. Each such variable assignment is performed just before the processing of the following *file*, if any. Thus, an assignment before the first *file* argument is executed after the **BEGIN** actions (if any), while an assignment after the last *file* argument is executed before the **END** actions (if any). If there are no *file* arguments, assignments are executed before processing the standard input.

INPUT FILES

Input files to the **nawk** program from any of the following sources:

- any *file* operands or their equivalents, achieved by modifying the **nawk** variables **ARGV** and **ARGC**
- standard input in the absence of any *file* operands
- arguments to the **getline** function

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

If −**f** *progfile* is specified, the files named by each of the *progfile* option-arguments must be text files containing an **nawk** program.

The standard input are used only if no *file* operands are specified, or if a *file* operand is −.

EXTENDED DESCRIPTION

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

pattern { *action* }

Either the pattern or the action (including the enclosing brace characters) can be omitted. Pattern-action statements are separated by a semicolon or by a newline.

A missing pattern matches any record of input, and a missing action is equivalent to an action that writes the matched record of input to standard output.

Execution of the **nawk** program starts 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) is processed by reading data from the file until a record separator is seen (a newline character by default), splitting the current record into fields using the current value of **FS,** 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 is executed before evaluating subsequent patterns. Last, the actions associated with all **END** patterns is executed in the order they occur in the program.

Expressions in nawk

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

Syntax	Name	Type of Result	Associativity
(*expr*)	Grouping	type of *expr*	n/a
$*expr*	Field reference	string	n/a
++ *lvalue*	Pre-increment	numeric	n/a
−− *lvalue*	Pre-decrement	numeric	n/a
lvalue ++	Post-increment	numeric	n/a
lvalue −−	Post-decrement	numeric	n/a
expr ^ *expr*	Exponentiation	numeric	right
! *expr*	Logical not	numeric	n/a
+ *expr*	Unary plus	numeric	n/a
− *expr*	Unary minus	numeric	n/a
expr * *expr*	Multiplication	numeric	left
expr / *expr*	Division	numeric	left
expr % *expr*	Modulus	numeric	left
expr + *expr*	Addition	numeric	left

expr – *expr*	Subtraction	numeric	left
expr expr	String concatenation	string	left
expr < *expr*	Less than	numeric	none
expr <= *expr*	Less than or equal to	numeric	none
expr != *expr*	Not equal to	numeric	none
expr == *expr*	Equal to	numeric	none
expr > *expr*	Greater than	numeric	none
expr >= *expr*	Greater than or equal to	numeric	none
expr ~ *expr*	ERE match	numeric	none
expr !~ *expr*	ERE non-match	numeric	none
expr in array	Array membership	numeric	left
(*index*) in *array*	Multi-dimension array membership	numeric	left
expr && *expr*	Logical AND	numeric	left
expr || *expr*	Logical OR	numeric	left
expr1 ? *expr2* : *expr3*	Conditional expression	type of selected *expr2* or *expr3*	right
lvalue ^= *expr*	Exponentiation assignment	numeric	right
lvalue %= *expr*	Modulus assignment	numeric	right
lvalue *= *expr*	Multiplication assignment	numeric	right
lvalue /= *expr*	Division assignment	numeric	right
lvalue += *expr*	Addition assignment	numeric	right
lvalue –= *expr*	Subtraction assignment	numeric	right
lvalue = *expr*	Assignment	type of *expr*	right

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

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

A numeric value that is exactly equal to the value of an integer is converted to a string by the equivalent of a call to the **sprintf** function 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 is 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 document 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 is considered to be a *numeric string* in the following case:

1. Any leading and trailing blank characters is ignored.
2. If the first unignored character is a **+** or −, it is ignored.
3. If the remaining unignored characters would be lexically recognized as a **NUMBER** token, the string is considered a *numeric string*.

If a − character is ignored in the above steps, the numeric value of the *numeric string* is the negation of the numeric value of the recognized **NUMBER** token. Otherwise the numeric value of the *numeric string* is the numeric value of the recognized **NUMBER** token. Whether or not a string is a *numeric string* is relevant only in contexts where that term is used in this section.

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

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

The **nawk** language supplies arrays that are used for storing numbers or strings. Arrays need not be declared. They are initially empty, and their sizes changes 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 as an expression, as described in the grammar. Unsubscripted array names are 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**.

A valid array *index* consists of one or more comma-separated expressions, similar to the way in which multi-dimensional arrays are indexed in some programming languages. Because **nawk** arrays are really one dimensional, such a comma-separated list is 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 are equivalent:

```
var[expr1, expr2, ... exprn]
var[expr1 SUBSEP expr2 SUBSEP ... SUBSEP exprn]
```

A multi-dimensioned *index* used with the **in** operator must be put in parentheses. The **in** operator, which tests for the existence of a particular array element, does not create the element if it does not exist. Any other reference to a non-existent array element automatically creates it.

Variables and Special Variables

Variables can be used in an **nawk** program by referencing them. With the exception of function parameters, 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 are designated by a **$** followed by a number or numerical expression. The effect of the field number *expression* evaluating to anything other than a non-negative integer is unspecified; uninitialized variables or string values need not be converted to numeric values in this context. New field variables are created by assigning a value to them. References to non-existent fields (that is, fields after **$NF**) produce the null string. However, assigning to a non-existent field (for example, **$(NF+2) = 5**) increases 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 has a string value when created. If the string, with any occurrence of the decimal-point character from the current locale changed to a period character, is considered a *numeric string* (see **Expressions in nawk** above), the field variable also has the numeric value of the *numeric string*.

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

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, **nawk** treats the next non-null element of **ARGV**, up to the current value of **ARGC**−1, inclusive, as the name of the next input file. Setting an element of **ARGV** to null means that it is not treated as an input file. The name – indicates the standard input. If an argument matches the format of an *assignment* operand, this argument is treated as an assignment rather than a *file* argument.

/usr/xcu4/bin/awk

CONVFM 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. The indices of the array are strings consisting of the names of the environment variables, and the value of each array element is a string consisting of the value of that variable. If the value of an environment variable is considered a *numeric string*, the array element also has its numeric value.

In all cases where **nawk** behavior is affected by environment variables (including the environment of any commands that **nawk** executes via the **system** function or via pipeline redirections with the **print** statement, the **printf** statement, or the **getline** function), the environment used is the environment at the time **nawk** began executing; it is implementation-dependent 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; a space character 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** retains 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 **""** **"%.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 separator; a space character by default.

ORS The **print** output record separator; a newline character by default.

LENGTH 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; a newline character 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 produce empty records at the beginning or end of input, and the field separator is always newline, no matter what the value of **FS**.

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 multi-dimensional arrays; the default value is implementation-dependent.

Regular Expressions

The **nawk** utility makes use of the extended regular expression notation (see **regex**(5)) except that it allows the use of C-language conventions to escape special characters within the EREs, namely \\, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**, and those specified in the following table. These escape sequences are recognized both inside and outside bracket expressions. Note that records need not be separated by newline characters and string constants can contain newline characters, so even the **\n** sequence is valid in **nawk** EREs. Using a slash character within the regular expression requires escaping as shown in the table below:

Escape Sequence	Description	Meaning
\"	Backslash quotation-mark	Quotation-mark character
\/	Backslash slash	Slash character
ddd	A backslash character followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0, (that is, representation of the NULL character), the behavior is undefined.	The character encoded by the one-, two- or three-digit octal integer. Multi-byte characters require multiple, concatenated escape sequences, including the leading \ for each byte.
c	A backslash character followed by any character not described in this table or special characters (\\\\, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**).	Undefined

A regular expression can be matched against a specific field or string by using one of the two regular expression matching operators, **˜** and **!˜**. These operators 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 evaluates to the value **1**, and the **!˜** expression evaluates to the value **0**. If the regular expression does not match the string, the **˜** expression evaluates to the value **0**, and the **!˜** expression evaluates to the value **1**. If the right-hand operand is any expression other than the lexical token **ERE**, the string value of the expression is interpreted as an extended regular expression, including the escape conventions described above. Note that these same escape conventions also are applied in the determining the value of a string literal (the lexical token **STRING**), and is 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 is the equivalent of:

$0 ˜ /*ere***/**

The *ere* argument to the **gsub, match, sub** functions, and the *fs* argument to the **split** function (see **String Functions**) is interpreted as extended regular expressions. These can be either **ERE** tokens or arbitrary expressions, and are 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 is a single space character. The following describes **FS** behavior:

1. If **FS** is a single character:
   a. If **FS** is the space character, skip leading and trailing blank characters; fields are delimited by sets of one or more blank characters.
   b. Otherwise, if **FS** is any other character *c*, fields are delimited by each single occurrence of *c*.

2. Otherwise, the string value of **FS** is considered to be an extended regular expression. Each occurrence of a sequence matching the extended regular expression delimits fields.

Except in the **gsub, match, split,** and **sub** built-in functions, regular expression matching is based on input records; that is, record separator characters (the first character of the value of the variable **RS**, a newline character by default) cannot be embedded in the expression, and no expression matches the record separator character. If the record separator is not a newline character, newline characters embedded in the expression can be matched. In those four built-in functions, regular expression matching are based on text strings. So, any character (including the newline character and the record separator) can be embedded in the pattern and an appropriate pattern will match any character. However, in all **nawk** regular expression matching, the use of one or more NUL characters in the pattern, input record or text string produces undefined results.

Patterns

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

Special Patterns

The **nawk** utility recognizes two special patterns, **BEGIN** and **END**. Each **BEGIN** pattern is matched once and its associated action executed before the first record of input is read (except possibly by use of the **getline** function in a prior **BEGIN** action) and before command line assignment is done. Each **END** pattern is matched once and its associated action executed after the last record of input has been read. These two patterns have associated actions.

BEGIN and **END** do not combine with other patterns. Multiple **BEGIN** and **END** patterns are allowed. The actions associated with the **BEGIN** patterns are 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 **nawk** program consists of only actions with the pattern **BEGIN**, and the **BEGIN** action contains no **getline** function, **nawk** exits without reading its input when the last statement in the last **BEGIN** action is executed. If an **nawk** program consists of only actions with the pattern **END** or only actions with the patterns **BEGIN** and **END**, the input is read before the statements in the **END** actions are executed.

Expression Patterns

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

Pattern Ranges

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

Actions

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

if (*expression*) *statement* [**else** *statement*]
while (*expression*) *statement*
do *statement* **while** (*expression*)
for (*expression* ; *expression* ; *expression*) *statement*
for (*var* **in** *array*) *statement*
delete *array*[*subscript*] #delete an array element
break
continue
{ [*statement*] . . . }
expression # commonly variable = expression
print [*expression-list*] [>*expression*]
printf *format* [, *expression-list*] [>*expression*]
next # skip remaining patterns on this input line
exit [**expr**] # skip the rest of the input; exit status is expr
return [**expr**]

Any single statement can be replaced by a statement list enclosed in braces. The statements are terminated by newline characters or semicolons, and are executed sequentially in the order that they appear.

The **next** statement causes 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 invokes 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 terminates the program without further execution of **END** actions. If an expression is specified in an **exit** statement, its numeric value is the exit status of **nawk**, unless subsequent errors are encountered or a subsequent **exit** statement with an expression is executed.

Output Statements

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

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

In all cases, the *expression* is 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 is opened, creating it if necessary and using the first form, truncating the file. The output then is appended to the file. As long as the file remains open, subsequent calls in which *expression* evaluates to the same string value simply appends output to the file. The file remains open until the **close** function, which is called with an expression that evaluates to the same string value.

The third form writes output onto a stream piped to the input of a command. The stream is created if no stream is currently open with the value of *expression* as its command name. The stream created is equivalent to one created by a call to the **popen**(3S) function 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 writes output to the existing stream. The stream will remain open until the **close** function is called with an expression that evaluates to the same string value. At that time, the stream is closed as if by a call to the **pclose** function.

These output statements 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 writes 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 is taken as strings, being converted if necessary; with the exception that the **printf** format in **OFMT** is used instead of the value in **CONVFMT**. An empty expression list stands for the whole input record **(**$0**)**.

The **printf** statement produces output based on a notation similar to the File Format Notation used to describe file formats in this document Output is produced as specified with the first expression argument as the string **format** and subsequent expression arguments as the strings **arg1** to **argn,** inclusive, with the following exceptions:

1. The *format* is 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, is 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 is treated as an ordinary character that is copied to the output.

3. The *escape sequences* beginning with a backslash character is treated as sequences of ordinary characters that are copied to the output. Note that these same sequences is interpreted lexically by **nawk** when they appear in literal strings, but they is not 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 is fetched and its numeric value taken as the field width or precision.

5. The implementation does not precede or follow output from the **d** or **u** conversion specifications with blank characters not specified by the *format* string.

6. The implementation does not precede output from the **o** conversion specification with leading zeros 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 is 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 will 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 will be evaluated. With the exception of the **c** conversion, the value will be converted to the appropriate type for the conversion specification.

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

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

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

Functions

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

Arithmetic Functions

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

atan2(*y*,*x***)** Return arctangent of *y* / *x*.

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 will be truncated toward 0 when *x* **>** 0.

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

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

String Functions

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

gsub(*ere*, *repl*[, *in*])
Behave like **sub** (see below), except that it will 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** will be set to the starting position (which is the same as the returned value), zero if no match is found; **RLENGTH** will be set to the length of the matched string, −1 if no match is found.

split(*s*, *a*[,*fs*]) Split the string *s* into array elements *a*[1], *a*[2], **...,** *a*[*n*], and return *n*. The separation will be done with the extended regular expression *fs* or with the field separator **FS** if *fs* is not given. Each array element will 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 character, would be considered a *numeric string*; the array element will 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* will be replaced by the string from *in* that matches the regular expression. For each occurrence of backslash (\) encountered when scanning the string *repl* from beginning to end, the next character is taken literally and loses its special meaning (for example, \& will be interpreted as a literal ampersand character). Except for & and \, it is unspecified what the special meaning of any such character is. If *in* is specified and it is not an *lvalue* the behavior is undefined. If *in* is omitted, **nawk** will 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 will 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 upper-case letter specified to have a **tolower** mapping by the **LC_CTYPE** category of the current locale will be replaced in the returned string by the lower-case letter specified by the mapping. Other characters in *s* will be unchanged in the returned string.

toupper(*s*) Return a string based on the string *s.* Each character in *s* that is a lower-case letter specified to have a **toupper** mapping by the **LC_CTYPE** category of the current locale will be replaced in the returned string by the upper-case letter specified by the mapping. Other characters in *s* will 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 below.

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-dependent. If the close was successful, the function will return zero; otherwise, it will return non-zero.

expression | getline [*var*]
Read a record of input from a stream piped from the output of a command. The stream will be created if no stream is currently open with the value of *expression* as its command name. The stream created will 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 will read subsequent records from the file. The stream will remain open until the **close** function is called with an expression that evaluates to the same string value. At that time, the stream will be closed as if by a call to the **pclose** function. If *var* is missing, **$0** and **NF** will be set; otherwise, *var* will be set.

The **getline** operator can form ambiguous constructs when there are operators that are not in parentheses (including concatenate) to the left of the **|** (to the beginning of the expression containing **getline).** In the context of the **$** operator, **|** behaves as if it had a lower precedence than **$.** The result of evaluating other operators is unspecified, and all such uses of portable applications must be put in parentheses properly.

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

getline [*var*] **<** *expression*
Read the next record of input from a named file. The *expression* will be evaluated to produce a string that is used as a full pathname. If the file of that name is not currently open, it will be opened. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value will read subsequent records from the file. The file will 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** will be set; otherwise, *var* will be set.

The **getline** operator can form ambiguous constructs when there are binary operators that are not in parentheses (including concatenate) to the right of the < (up to the end of the expression containing the **getline**). The result of evaluating such a construct is unspecified, and all such uses of portable applications must be put in parentheses properly.

system(*expression*)
Execute the command given by *expression* in a manner equivalent to the **system**(3S) function and return the exit status of the command.

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

Where strings are used as the name of a file or pipeline, the strings must be textually identical. The terminology ‘‘same string value’’ implies that ‘‘equivalent strings’’, even those that differ only by space characters, represent different files.

User-defined Functions

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

function *name*(*args*, . . .) { *statements* }

A function can be referred to anywhere in an **nawk** program; in particular, its use can precede its definition. The scope of a function will 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 will be passed by value if scalar and by reference if array name. Argument names will be local to the function; all other variable names will be global. The same name will not be used as both an argument name and as the name of a function or a special **nawk** variable. The same name must not be used both as a variable name with global scope and as the name of a function. The same name must 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 will 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 will 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. Function calls can be nested and recursive calls can be made upon functions. Upon return from any nested or recursive function call, the values of all of the calling function’s parameters will 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 characters are 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.

USAGE

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

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

EXAMPLES

The **nawk** program specified in the command line is most easily specified within single-quotes (for example, '*program*') for applications using **sh**, because **nawk** programs commonly contain characters that are special to the shell, including double-quotes. In the cases where a **nawk** 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 **nawk** programs:

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

   ```
   $3 > 5
   ```

2. Write every tenth line:

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

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

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

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

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

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

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

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

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

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

   ```
   $2 ~ "\\\\"
   ```

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

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

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

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

10. Write lines longer than 72 characters:

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

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

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

12. Same, with input fields separated by comma or space and tab characters, or both:

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

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

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

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

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

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

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

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

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

17. Simulate **echo**:

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

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

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

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

```
Page#
```

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

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

then the command line:

nawk −f program n=5 input

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

ENVIRONMENT

See **environ**(5) for descriptions of the following environment variables that affect execution: **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_NUMERIC**, and **NLSPATH**.

EXIT STATUS

The following exit values are returned:

0 All input files were processed successfully.

>0 An error occurred.

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

SEE ALSO

awk(1), **ed**(1), **egrep**(1), **grep**(1), **lex**(1), **sed**(1), **popen**(3S), **printf**(3S), **environ**(5), **regex**(5)

The **awk** chapter in the *Solaris Advanced User's Guide*.

A. V. Aho, B. W. Kerninghan, P. J. Weinberger, *The AWK Programming Language* Addison-Wesley, 1988.

DIAGNOSTICS

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

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

NOTES

nawk is a new version of **awk** that provides capabilities unavailable in previous versions. This version will become the default version of **awk** in the next major release.

Input white space is not preserved on output if fields are involved.

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string (**""**) to it.