

MAWK(1)

USER COMMANDS

MAWK(1)

#### NAME

mawk - pattern scanning and text processing language

#### SYNOPSIS

```
mawk [-W option] [-F value] [-v var=value] [--] 'program text' [file ...]
mawk [-W option] [-F value] [-v var=value] [-f program-file] [--] [file ...]
```

#### DESCRIPTION

mawk is an interpreter for the AWK Programming Language. The AWK language is useful for manipulation of data files, text retrieval and processing, and for prototyping and experimenting with algorithms. mawk is a new awk meaning it implements the AWK language as defined in Aho, Kernighan and Weinberger, *The AWK Programming Language*, Addison-Wesley Publishing, 1988. (Hereafter referred to as the AWK book.) mawk conforms to the Posix 1003.2 (draft 11.3) definition of the AWK language which contains a few features not described in the AWK book, and mawk provides a small number of extensions.

An AWK program is a sequence of pattern {action} pairs and function definitions. Short programs are entered on the command line usually enclosed in ' ' to avoid shell interpretation. Longer programs can be read in from a file with the -f option. Data input is read from the list of files on the command line or from standard input when the list is empty. The input is broken into records as determined by the record separator variable, RS. Initially, RS = "\n" and records are synonymous with lines. Each record is compared against each pattern and if it matches, the program text for {action} is executed.

#### OPTIONS

- F value sets the field separator, FS, to value.
- f file Program text is read from file instead of from the command line. Multiple -f options are allowed.
- v var=value assigns value to program variable var.
- indicates the unambiguous end of options.

The above options will be available with any Posix compatible implementation of AWK, and implementation specific options are prefaced with -W. mawk provides six:

- W version mawk writes its version and copyright to stdout and compiled limits to stderr and exits 0.
- W dump writes an assembler like listing of the internal representation of the program to stdout and exits 0 (on successful compilation).
- W interactive sets unbuffered writes to stdout and line buffered reads from stdin. Records from stdin are lines regardless of the value of RS.
- W exec file Program text is read from file and this is the last option. Useful on systems that support the #! "magic number" convention for executable scripts.
- W sprintf=num adjusts the size of mawk's internal sprintf buffer to num bytes. More than rare use of this option indicates mawk should be recompiled.

-W posix\_space forces mawk not to consider '\n' to be space.

The short forms -W[vdiesp] are recognized and on some systems -We is mandatory to avoid command line length limitations.

## THE AWK LANGUAGE

### 1. Program structure

An AWK program is a sequence of pattern {action} pairs and user function definitions.

A pattern can be:

```
BEGIN
END
expression
expression , expression
```

One, but not both, of pattern {action} can be omitted. If {action} is omitted it is implicitly { print }. If pattern is omitted, then it is implicitly matched. BEGIN and END patterns require an action.

Statements are terminated by newlines, semi-colons or both. Groups of statements such as actions or loop bodies are blocked via { ... } as in C. The last statement in a block doesn't need a terminator. Blank lines have no meaning; an empty statement is terminated with a semi-colon. Long statements can be continued with a backslash, \. A statement can be broken without a backslash after a comma, left brace, &&, ||, do, else, the right parenthesis of an if, while or for statement, and the right parenthesis of a function definition. A comment starts with # and extends to, but does not include the end of line.

The following statements control program flow inside blocks.

```
if ( expr ) statement
if ( expr ) statement else statement
while ( expr ) statement
do statement while ( expr )
for ( opt_expr ; opt_expr ; opt_expr ) statement
for ( var in array ) statement
continue
break
```

### 2. Data types, conversion and comparison

There are two basic data types, numeric and string. Numeric constants can be integer like -2, decimal like 1.08, or in scientific notation like -1.1e4 or .28E-3. All numbers are represented internally and all computations are done in floating point arithmetic. So for example, the expression 0.2e2 == 20 is true and true is represented as 1.0.

String constants are enclosed in double quotes.

```
"This is a string with a newline at the end.\n"
```

Strings can be continued across a line by escaping (\) the newline. The following escape sequences are recognized.

```
\\      \
\"      "
\a      alert, ascii 7
```

```
\b      backspace, ascii 8
\t      tab, ascii 9
\n      newline, ascii 10
\v      vertical tab, ascii 11
\f      formfeed, ascii 12
\r      carriage return, ascii 13
\ddd    1, 2 or 3 octal digits for ascii ddd
\xhh    1 or 2 hex digits for ascii hh
```

If you escape any other character `\c`, you get `\c`, i.e., `mawk` ignores the escape.

There are really three basic data types; the third is number and string which has both a numeric value and a string value at the same time. User defined variables come into existence when first referenced and are initialized to null, a number and string value which has numeric value 0 and string value "". Non-trivial number and string typed data come from input and are typically stored in fields. (See section 4).

The type of an expression is determined by its context and automatic type conversion occurs if needed. For example, to evaluate the statements

```
y = x + 2 ; z = x "hello"
```

The value stored in variable `y` will be typed numeric. If `x` is not numeric, the value read from `x` is converted to numeric before it is added to 2 and stored in `y`. The value stored in variable `z` will be typed string, and the value of `x` will be converted to string if necessary and concatenated with "hello". (Of course, the value and type stored in `x` is not changed by any conversions.) A string expression is converted to numeric using its longest numeric prefix as with `atof(3)`. A numeric expression is converted to string by replacing `expr` with `sprintf(CONVFMT, expr)`, unless `expr` can be represented on the host machine as an exact integer then it is converted to `sprintf("%d", expr)`. `Sprintf()` is an AWK built-in that duplicates the functionality of `sprintf(3)`, and `CONVFMT` is a built-in variable used for internal conversion from number to string and initialized to `"%.6g"`. Explicit type conversions can be forced, `expr ""` is string and `expr+0` is numeric.

To evaluate, `expr1 rel-op expr2`, if both operands are numeric or number and string then the comparison is numeric; if both operands are string the comparison is string; if one operand is string, the non-string operand is converted and the comparison is string. The result is numeric, 1 or 0.

In boolean contexts such as, `if ( expr )` statement, a string expression evaluates true if and only if it is not the empty string ""; numeric values if and only if not numerically zero.

### 3. Regular expressions

In the AWK language, records, fields and strings are often tested for matching a regular expression. Regular expressions are enclosed in slashes, and

```
expr ~ /r/
```

is an AWK expression that evaluates to 1 if `expr` "matches" `r`, which means a substring of `expr` is in the set of strings defined by `r`. With no match the expression evaluates to 0; replacing `~` with the "not match" operator, `!~`, reverses the meaning. As pattern-action pairs,

```
/r/ { action } and $0 ~ /r/ { action }
```

are the same, and for each input record that matches `r`, `action` is

executed. In fact, `/r/` is an AWK expression that is equivalent to `($0 ~ /r/)` anywhere except when on the right side of a match operator or passed as an argument to a built-in function that expects a regular expression argument.

AWK uses extended regular expressions as with `egrep(1)`. The regular expression metacharacters, i.e., those with special meaning in regular expressions are

```
^ $ . [ ] | ( ) * + ?
```

Regular expressions are built up from characters as follows:

<code>c</code>	matches any non-metacharacter <code>c</code> .
<code>\c</code>	matches a character defined by the same escape sequences used in string constants or the literal character <code>c</code> if <code>\c</code> is not an escape sequence.
<code>.</code>	matches any character (including newline).
<code>^</code>	matches the front of a string.
<code>\$</code>	matches the back of a string.
<code>[c1c2c3...]</code>	matches any character in the class <code>c1c2c3...</code> . An interval of characters is denoted <code>c1-c2</code> inside a class <code>[...]</code> .
<code>[^c1c2c3...]</code>	matches any character not in the class <code>c1c2c3...</code> .

Regular expressions are built up from other regular expressions as follows:

<code>r1r2</code>	matches <code>r1</code> followed immediately by <code>r2</code> (concatenation).
<code>r1   r2</code>	matches <code>r1</code> or <code>r2</code> (alternation).
<code>r*</code>	matches <code>r</code> repeated zero or more times.
<code>r+</code>	matches <code>r</code> repeated one or more times.
<code>r?</code>	matches <code>r</code> zero or once.
<code>(r)</code>	matches <code>r</code> , providing grouping.

The increasing precedence of operators is alternation, concatenation and unary (`*`, `+` or `?`).

For example,

```
 /^[_a-zA-Z][_a-zA-Z0-9]*$/ and
 /^[+-]?([0-9]+\.\.?|\.[0-9])[0-9]*([eE][+-]?[0-9]+)?$/
```

are matched by AWK identifiers and AWK numeric constants respectively. Note that `.` has to be escaped to be recognized as a decimal point, and that metacharacters are not special inside character classes.

Any expression can be used on the right hand side of the `~` or `!~` operators or passed to a built-in that expects a regular expression. If needed, it is converted to string, and then interpreted as a regular expression. For example,

```
BEGIN { identifier = "[_a-zA-Z][_a-zA-Z0-9]*" }
```

```
$0 ~ "^" identifier
```

prints all lines that start with an AWK identifier.

mawk recognizes the empty regular expression, `//`, which matches the empty string and hence is matched by any string at the front, back and between every character. For example,

```
echo abc | mawk { gsub(//, "X") ; print }
XaXbXcX
```

#### 4. Records and fields

Records are read in one at a time, and stored in the field variable `$0`. The record is split into fields which are stored in `$1`, `$2`, ..., `$NF`. The built-in variable `NF` is set to the number of fields, and `NR` and `FNR` are incremented by 1. Fields above `$NF` are set to "".

Assignment to `$0` causes the fields and `NF` to be recomputed. Assignment to `NF` or to a field causes `$0` to be reconstructed by concatenating the `$i`'s separated by `OFS`. Assignment to a field with index greater than `NF`, increases `NF` and causes `$0` to be reconstructed.

Data input stored in fields is string, unless the entire field has numeric form and then the type is number and string. For example,

```
echo 24 24E |
mawk '{ print($1>100, $1>"100", $2>100, $2>"100") }'
0 1 1 1
```

`$0` and `$2` are string and `$1` is number and string. The first comparison is numeric, the second is string, the third is string (100 is converted to "100"), and the last is string.

#### 5. Expressions and operators

The expression syntax is similar to C. Primary expressions are numeric constants, string constants, variables, fields, arrays and function calls. The identifier for a variable, array or function can be a sequence of letters, digits and underscores, that does not start with a digit. Variables are not declared; they exist when first referenced and are initialized to null.

New expressions are composed with the following operators in order of increasing precedence.

assignment	= += -= *= /= %= ^=
conditional	? :
logical or	
logical and	&&
array membership	in
matching	~ !~
relational	< > <= >= == !=
concatenation	(no explicit operator)
add ops	+ -
mul ops	* / %
unary	+ -
logical not	!
exponentiation	^
inc and dec	++ -- (both post and pre)
field	\$

Assignment, conditional and exponentiation associate right to left; the other operators associate left to right. Any expression can be parenthesized.

#### 6. Arrays

Awk provides one-dimensional arrays. Array elements are expressed as `array[expr]`. `Expr` is internally converted to string type, so, for example, `A[1]` and `A["1"]` are the same element and the actual index is "1". Arrays indexed by strings are called associative arrays. Initially an array is empty; elements exist when first accessed. An expression, `expr` in array evaluates to 1 if `array[expr]` exists, else to 0.

There is a form of the `for` statement that loops over each index of an array.

```
for ( var in array ) statement
```

sets `var` to each index of array and executes statement. The order that `var` transverses the indices of array is not defined.

The statement, `delete array[expr]`, causes `array[expr]` not to exist. `mawk` supports an extension, `delete array`, which deletes all elements of array.

Multidimensional arrays are synthesized with concatenation using the built-in variable `SUBSEP`. `array[expr1,expr2]` is equivalent to `array[expr1 SUBSEP expr2]`. Testing for a multidimensional element uses a parenthesized index, such as

```
if ( (i, j) in A ) print A[i, j]
```

## 7. Builtin-variables

The following variables are built-in and initialized before program execution.

<code>ARGC</code>	number of command line arguments.
<code>ARGV</code>	array of command line arguments, 0.. <code>ARGC</code> -1.
<code>CONVFMT</code>	format for internal conversion of numbers to string, initially = <code>"%.6g"</code> .
<code>ENVIRON</code>	array indexed by environment variables. An environment string, <code>var=value</code> is stored as <code>ENVIRON[var] = value</code> .
<code>FILENAME</code>	name of the current input file.
<code>FNR</code>	current record number in <code>FILENAME</code> .
<code>FS</code>	splits records into fields as a regular expression.
<code>NF</code>	number of fields in the current record.
<code>NR</code>	current record number in the total input stream.
<code>OFMT</code>	format for printing numbers; initially = <code>"%.6g"</code> .
<code>OFS</code>	inserted between fields on output, initially = " ".
<code>ORS</code>	terminates each record on output, initially = <code>"\n"</code> .
<code>RLENGTH</code>	length set by the last call to the built-in function, <code>match()</code> .
<code>RS</code>	input record separator, initially = <code>"\n"</code> .
<code>RSTART</code>	index set by the last call to <code>match()</code> .

SUBSEP      used to build multiple array subscripts, initially =  
              "\034".

## 8. Built-in functions

### String functions

`gsub(r,s,t) gsub(r,s)`  
Global substitution, every match of regular expression `r` in variable `t` is replaced by string `s`. The number of replacements is returned. If `t` is omitted, `$0` is used. An `&` in the replacement string `s` is replaced by the matched substring of `t`. `&` and `\&` put literal `&` and `\`, respectively, in the replacement string.

`index(s,t)`  
If `t` is a substring of `s`, then the position where `t` starts is returned, else 0 is returned. The first character of `s` is in position 1.

`length(s)`  
Returns the length of string `s`.

`match(s,r)`  
Returns the index of the first longest match of regular expression `r` in string `s`. Returns 0 if no match. As a side effect, `RSTART` is set to the return value. `RLENGTH` is set to the length of the match or -1 if no match. If the empty string is matched, `RLENGTH` is set to 0, and 1 is returned if the match is at the front, and `length(s)+1` is returned if the match is at the back.

`split(s,A,r) split(s,A)`  
String `s` is split into fields by regular expression `r` and the fields are loaded into array `A`. The number of fields is returned. See section 11 below for more detail. If `r` is omitted, `FS` is used.

`sprintf(format,expr-list)`  
Returns a string constructed from `expr-list` according to `format`. See the description of `printf()` below.

`sub(r,s,t) sub(r,s)`  
Single substitution, same as `gsub()` except at most one substitution.

`substr(s,i,n) substr(s,i)`  
Returns the substring of string `s`, starting at index `i`, of length `n`. If `n` is omitted, the suffix of `s`, starting at `i` is returned.

`tolower(s)`  
Returns a copy of `s` with all upper case characters converted to lower case.

`toupper(s)`  
Returns a copy of `s` with all lower case characters converted to upper case.

### Arithmetic functions

`atan2(y,x)`      Arctan of `y/x` between  $-\pi$  and  $\pi$ .

`cos(x)`            Cosine function, `x` in radians.

`exp(x)`            Exponential function.

`int(x)` Returns `x` truncated towards zero.

`log(x)` Natural logarithm.

`rand()` Returns a random number between zero and one.

`sin(x)` Sine function, `x` in radians.

`sqrt(x)` Returns square root of `x`.

`srand(expr) srand()`  
Seeds the random number generator, using the clock if `expr` is omitted, and returns the value of the previous seed. `mawk` seeds the random number generator from the clock at startup so there is no real need to call `srand()`. `Srand(expr)` is useful for repeating pseudo random sequences.

## 9. Input and output

There are two output statements, `print` and `printf`.

`print` writes `$0` ORS to standard output.

`print expr1, expr2, ..., exprn`  
writes `expr1` OFS `expr2` OFS ... `exprn` ORS to standard output. Numeric expressions are converted to string with OFMT.

`printf format, expr-list`  
duplicates the `printf` C library function writing to standard output. The complete ANSI C format specifications are recognized with conversions `%c`, `%d`, `%e`, `%E`, `%f`, `%g`, `%G`, `%i`, `%o`, `%s`, `%u`, `%x`, `%X` and `%%`, and conversion qualifiers `h` and `l`.

The argument list to `print` or `printf` can optionally be enclosed in parentheses. `Print` formats numbers using OFMT or `"%d"` for exact integers. `"%c"` with a numeric argument prints the corresponding 8 bit character, with a string argument it prints the first character of the string. The output of `print` and `printf` can be redirected to a file or command by appending `> file`, `>> file` or `| command` to the end of the print statement. Redirection opens file or command only once, subsequent redirections append to the already open stream. By convention, `mawk` associates the filename `"/dev/stderr"` with `stderr` which allows `print` and `printf` to be redirected to `stderr`. `mawk` also associates `"-"` and `"/dev/stdout"` with `stdin` and `stdout` which allows these streams to be passed to functions.

The input function `getline` has the following variations.

`getline`  
reads into `$0`, updates the fields, `NF`, `NR` and `FNR`.

`getline < file`  
reads into `$0` from `file`, updates the fields and `NF`.

`getline var`  
reads the next record into `var`, updates `NR` and `FNR`.

`getline var < file`  
reads the next record of `file` into `var`.

`command | getline`  
pipes a record from `command` into `$0` and updates the fields and `NF`.

```
command | getline var
      pipes a record from command into var.
```

Getline returns 0 on end-of-file, -1 on error, otherwise 1.

Commands on the end of pipes are executed by /bin/sh.

The function close(expr) closes the file or pipe associated with expr. Close returns 0 if expr is an open file, the exit status if expr is a piped command, and -1 otherwise. Close is used to reread a file or command, make sure the other end of an output pipe is finished or conserve file resources.

The function fflush(expr) flushes the output file or pipe associated with expr. Fflush returns 0 if expr is an open output stream else -1. Fflush without an argument flushes stdout. Fflush with an empty argument ("") flushes all open output.

The function system(expr) uses /bin/sh to execute expr and returns the exit status of the command expr. Changes made to the ENVIRON array are not passed to commands executed with system or pipes.

#### 10. User defined functions

The syntax for a user defined function is

```
function name( args ) { statements }
```

The function body can contain a return statement

```
return opt_expr
```

A return statement is not required. Function calls may be nested or recursive. Functions are passed expressions by value and arrays by reference. Extra arguments serve as local variables and are initialized to null. For example, csplit(s,A) puts each character of s into array A and returns the length of s.

```
function csplit(s, A, n, i)
{
  n = length(s)
  for( i = 1 ; i <= n ; i++ ) A[i] = substr(s, i, 1)
  return n
}
```

Putting extra space between passed arguments and local variables is conventional. Functions can be referenced before they are defined, but the function name and the '(' of the arguments must touch to avoid confusion with concatenation.

#### 11. Splitting strings, records and files

Awk programs use the same algorithm to split strings into arrays with split(), and records into fields on FS. mawk uses essentially the same algorithm to split files into records on RS.

Split(expr,A,sep) works as follows:

- (1) If sep is omitted, it is replaced by FS. Sep can be an expression or regular expression. If it is an expression of non-string type, it is converted to string.
- (2) If sep = " " (a single space), then <SPACE> is trimmed from the front and back of expr, and sep becomes <SPACE>. mawk defines <SPACE> as the regular expression /[\t\n]+/. Otherwise sep is treated as a regular expression, except that meta-characters are ignored for a string of length 1, e.g., split(x, A, "\*") and split(x,

A, /\\*/) are the same.

- (3) If `expr` is not string, it is converted to string. If `expr` is then the empty string "", `split()` returns 0 and `A` is set empty. Otherwise, all non-overlapping, non-null and longest matches of `sep` in `expr`, separate `expr` into fields which are loaded into `A`. The fields are placed in `A[1]`, `A[2]`, ..., `A[n]` and `split()` returns `n`, the number of fields which is the number of matches plus one. Data placed in `A` that looks numeric is typed number and string.

Splitting records into fields works the same except the pieces are loaded into `$1`, `$2`, ..., `$NF`. If `$0` is empty, `NF` is set to 0 and all `$i` to "".

`mawk` splits files into records by the same algorithm, but with the slight difference that `RS` is really a terminator instead of a separator. (`ORS` is really a terminator too).

E.g., if `FS = ":+"` and `$0 = "a::b:"`, then `NF = 3` and `$1 = "a"`, `$2 = "b"` and `$3 = ""`, but if `"a::b:"` is the contents of an input file and `RS = ":+"`, then there are two records `"a"` and `"b"`.

`RS = " "` is not special.

If `FS = ""`, then `mawk` breaks the record into individual characters, and, similarly, `split(s,A,"")` places the individual characters of `s` into `A`.

## 12. Multi-line records

Since `mawk` interprets `RS` as a regular expression, multi-line records are easy. Setting `RS = "\n\n+"`, makes one or more blank lines separate records. If `FS = " "` (the default), then single newlines, by the rules for `<SPACE>` above, become space and single newlines are field separators.

For example, if a file is `"a b\nc\n\n"`, `RS = "\n\n+"` and `FS = " "`, then there is one record `"a b\nc"` with three fields `"a"`, `"b"` and `"c"`. Changing `FS = "\n"`, gives two fields `"a b"` and `"c"`; changing `FS = ""`, gives one field identical to the record.

If you want lines with spaces or tabs to be considered blank, set `RS = "\n([\t]*\n)+"`. For compatibility with other awks, setting `RS = ""` has the same effect as if blank lines are stripped from the front and back of files and then records are determined as if `RS = "\n\n+"`. Posix requires that `"\n"` always separates records when `RS = ""` regardless of the value of `FS`. `mawk` does not support this convention, because defining `"\n"` as `<SPACE>` makes it unnecessary.

Most of the time when you change `RS` for multi-line records, you will also want to change `ORS` to `"\n\n"` so the record spacing is preserved on output.

## 13. Program execution

This section describes the order of program execution. First `ARGC` is set to the total number of command line arguments passed to the execution phase of the program. `ARGV[0]` is set the name of the AWK interpreter and `ARGV[1]` ... `ARGV[ARGC-1]` holds the remaining command line arguments exclusive of options and program source. For example with

```
mawk -f prog v=1 A t=hello B
```

`ARGC = 5` with `ARGV[0] = "mawk"`, `ARGV[1] = "v=1"`, `ARGV[2] = "A"`, `ARGV[3] = "t=hello"` and `ARGV[4] = "B"`.

Next, each BEGIN block is executed in order. If the program consists entirely of BEGIN blocks, then execution terminates, else an input stream is opened and execution continues. If ARGV equals 1, the input stream is set to stdin, else the command line arguments ARGV[1] ... ARGV[ARGV-1] are examined for a file argument.

The command line arguments divide into three sets: file arguments, assignment arguments and empty strings "". An assignment has the form var=string. When an ARGV[i] is examined as a possible file argument, if it is empty it is skipped; if it is an assignment argument, the assignment to var takes place and i skips to the next argument; else ARGV[i] is opened for input. If it fails to open, execution terminates with exit code 2. If no command line argument is a file argument, then input comes from stdin. Getline in a BEGIN action opens input. "-" as a file argument denotes stdin.

Once an input stream is open, each input record is tested against each pattern, and if it matches, the associated action is executed. An expression pattern matches if it is boolean true (see the end of section 2). A BEGIN pattern matches before any input has been read, and an END pattern matches after all input has been read. A range pattern, expr1,expr2 , matches every record between the match of expr1 and the match expr2 inclusively.

When end of file occurs on the input stream, the remaining command line arguments are examined for a file argument, and if there is one it is opened, else the END pattern is considered matched and all END actions are executed.

In the example, the assignment v=1 takes place after the BEGIN actions are executed, and the data placed in v is typed number and string. Input is then read from file A. On end of file A, t is set to the string "hello", and B is opened for input. On end of file B, the END actions are executed.

Program flow at the pattern {action} level can be changed with the

```
next
exit opt_expr
```

statements. A next statement causes the next input record to be read and pattern testing to restart with the first pattern {action} pair in the program. An exit statement causes immediate execution of the END actions or program termination if there are none or if the exit occurs in an END action. The opt\_expr sets the exit value of the program unless overridden by a later exit or subsequent error.

#### EXAMPLES

1. emulate cat.

```
{ print }
```

2. emulate wc.

```
{ chars += length($0) + 1 # add one for the \n
  words += NF
}

END{ print NR, words, chars }
```

3. count the number of unique "real words".

```
BEGIN { FS = "[^A-Za-z]+" }

{ for(i = 1 ; i <= NF ; i++) word[$i] = "" }
```

```
END { delete word[""]
      for ( i in word ) cnt++
      print cnt
    }
```

4. sum the second field of every record based on the first field.

```
$1 ~ /credit|gain/ { sum += $2 }
$1 ~ /debit|loss/ { sum -= $2 }

END { print sum }
```

5. sort a file, comparing as string

```
{ line[NR] = $0 " " } # make sure of comparison type
                        # in case some lines look numeric

END { isort(line, NR)
      for(i = 1 ; i <= NR ; i++) print line[i]
    }

#insertion sort of A[1..n]
function isort( A, n, i, j, hold)
{
  for( i = 2 ; i <= n ; i++)
  {
    hold = A[j = i]
    while ( A[j-1] > hold )
    { j-- ; A[j+1] = A[j] }
    A[j] = hold
  }
  # sentinel A[0] = "" will be created if needed
}
```

#### COMPATIBILITY ISSUES

The Posix 1003.2(draft 11.3) definition of the AWK language is AWK as described in the AWK book with a few extensions that appeared in SystemVR4 nawk. The extensions are:

New functions: toupper() and tolower().

New variables: ENVIRON[] and CONVFMT.

ANSI C conversion specifications for printf() and sprintf().

New command options: -v var=value, multiple -f options and implementation options as arguments to -W.

Posix AWK is oriented to operate on files a line at a time. RS can be changed from "\n" to another single character, but it is hard to find any use for this -- there are no examples in the AWK book. By convention, RS = "", makes one or more blank lines separate records, allowing multi-line records. When RS = "", "\n" is always a field separator regardless of the value in FS.

mawk, on the other hand, allows RS to be a regular expression. When "\n" appears in records, it is treated as space, and FS always determines fields.

Removing the line at a time paradigm can make some programs simpler and can often improve performance. For example, redoing example 3 from above,

```
BEGIN { RS = "[^A-Za-z]+" }
```

```
{ word[ $0 ] = " " }

END { delete word[ " " ]
      for( i in word ) cnt++
      print cnt
}
```

counts the number of unique words by making each word a record. On moderate size files, mawk executes twice as fast, because of the simplified inner loop.

The following program replaces each comment by a single space in a C program file,

```
BEGIN {
  RS = "/\*([^\*]|\\*+[^\*])*\*/"
  # comment is record separator
  ORS = " "
  getline hold
}

{ print hold ; hold = $0 }

END { printf "%s" , hold }
```

Buffering one record is needed to avoid terminating the last record with a space.

With mawk, the following are all equivalent,

```
x ~ /\a\b/      x ~ "a\b"      x ~ "a\\b"
```

The strings get scanned twice, once as string and once as regular expression. On the string scan, mawk ignores the escape on non-escape characters while the AWK book advocates `\c` be recognized as `c` which necessitates the double escaping of meta-characters in strings. Posix explicitly declines to define the behavior which passively forces programs that must run under a variety of awks to use the more portable but less readable, double escape.

Posix AWK does not recognize `"/dev/std{out,err}"` or `\x` hex escape sequences in strings. Unlike ANSI C, mawk limits the number of digits that follows `\x` to two as the current implementation only supports 8 bit characters. The built-in `fflush` first appeared in a recent (1993) AT&T awk released to netlib, and is not part of the posix standard. Aggregate deletion with `delete array` is not part of the posix standard.

Posix explicitly leaves the behavior of `FS = ""` undefined, and mentions splitting the record into characters as a possible interpretation, but currently this use is not portable across implementations.

Finally, here is how mawk handles exceptional cases not discussed in the AWK book or the Posix draft. It is unsafe to assume consistency across awks and safe to skip to the next section.

```
substr(s, i, n) returns the characters of s in the intersection
of the closed interval [1, length(s)] and the half-open interval
[i, i+n). When this intersection is empty, the empty string is
returned; so substr("ABC", 1, 0) = "" and substr("ABC", -4, 6) =
"A".
```

Every string, including the empty string, matches the empty string at the front so, `s ~ //` and `s ~ ""`, are always 1 as is `match(s, //)` and `match(s, "")`. The last two set `RLENGTH` to 0.

`index(s, t)` is always the same as `match(s, t1)` where `t1` is the same as `t` with metacharacters escaped. Hence consistency with `match` requires that `index(s, "")` always returns 1. Also the condition, `index(s,t) != 0` if and only `t` is a substring of `s`, requires `index("", "") = 1`.

If `getline` encounters end of file, `getline var`, leaves `var` unchanged. Similarly, on entry to the `END` actions, `$0`, the fields and `NF` have their value unaltered from the last record.

SEE ALSO

`egrep(1)`

Aho, Kernighan and Weinberger, *The AWK Programming Language*, Addison-Wesley Publishing, 1988, (the AWK book), defines the language, opening with a tutorial and advancing to many interesting programs that delve into issues of software design and analysis relevant to programming in any language.

The *GAWK Manual*, The Free Software Foundation, 1991, is a tutorial and language reference that does not attempt the depth of the AWK book and assumes the reader may be a novice programmer. The section on AWK arrays is excellent. It also discusses Posix requirements for AWK.

BUGS

`mawk` cannot handle ascii NUL `\0` in the source or data files. You can output NUL using `printf` with `%c`, and any other 8 bit character is acceptable input.

`mawk` implements `printf()` and `sprintf()` using the C library functions, `printf` and `sprintf`, so full ANSI compatibility requires an ANSI C library. In practice this means the `h` conversion qualifier may not be available. Also `mawk` inherits any bugs or limitations of the library functions.

Implementors of the AWK language have shown a consistent lack of imagination when naming their programs.

AUTHOR

Mike Brennan ([brennan@whidbey.com](mailto:brennan@whidbey.com)).