Wmawk2

Version 2.0, September 22nd 2024

NAME

Wmawk2 - pattern scanning and text processing language

SYNOPSIS

Wmawk2 [-**W** *option*] [-**F** *value*] [-**v** *var=value*] [- -] 'program text' [file ...]
Wmawk2 [-**W** *option*] [-**F** *value*] [-**v** *var=value*] [-**f** *program-file*] [- -] [file ...]

DESCRIPTION

Wmawk2 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. **Wmawk2** 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.) **Wmawk2** 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 **Wmawk2** provides a small number of extensions.

Wmawk2 a port of mawk 1.9.9.6 to Windows. It was created because the version of on github (https://github.com/mikebrennan000/mawk-2) did not compile for Windows when using tdm-gcc 9.2.2, and when these issues were resolved the system command did not work. The systime() function was also added.

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. Note that “inner” quotes then need to be escaped as shown in the example below:

```
wmawk2 "BEGIN{print \"€\"}"
```

To get a \ it needs to be escaped as in:

```
wmawk2 "BEGIN {print \"\\ \"}"
```

which prints

```
\
```

Note there is a space after the third ‘\’ character.

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.

Also note that when reading text interactively from stdin the arrow keys, backspace and delete can be used to edit the line before pressing Enter. To signify EOF you need to type CONTROL-Z followed by Enter. Normally the CONTROL-Z character should be the first character on the line, if there are characters before the CONTROL-Z they will be passed to the awk script before EOF is detected. Any characters on a line after a CONTROL-Z are lost. There is a maximum line length on interactive input (8192 utf8 characters) – this is the only fixed limit in wmawk2. The command line to wmawk2 will be subject to limits depending on how it is called, but 32767 characters is believed to be the maximum possible in Windows 10 (and it may be significantly lower in some cases) – but this is not a limitation of wmawk2.

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**. **Wmawk2** provides seven:

-W version
Wmawk2 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.

-W sprintf=*num*
This option is obsolete, its left for backwards compatibility with mawk version 1.

-W traditional
This option disables Wmawk2 extensions (like systime()) which are not in the original mawk2. These are marked “`Wmawk2 extension” in this manual.`

-W posix_space
forces **Wmawk2** not to consider '\n' to be space.

The short forms **-W**[vdiestp] are recognized.

Note that the space after -W is optional, and upper and lower case are equivalent. Options can also be abbreviated so -W t, -Wt, -W T, -W trad, -W Traditional are all equivalent.

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. Wmawk2 version 2.0 and above also allow hexadecimal constants (only) in awk scripts prefixed by 0X or 0x, e.g. 0xff is equivalent to the decimal number 255. Earlier versions of Wmawk2 would consider numeric constants starting 0x or 0X to be equal to zero (so 0xff would be considered as 0). Octal constants are not supported by Wmawk2 (as this could change the meaning of some existing programs as 010 is 8 in octal, but 10 in decimal).

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., **Wmawk2** ignores the escape.

Note that in replacement strings \ escapes \ and \ escapes &, but only if the run of \ ends in & so that the script {sub(/B/,"\\\\") ; print} fed with the input ABC will give A\\C

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 the C function *atof*(). 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 the C function *sprintf*(), 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, *expr*1 **rel-op** *expr*2, 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 the UNIX/Linux command *egrep*. The regular expression metacharacters, i.e., those with special meaning in regular expressions are

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

Regular expressions are built up from characters as follows:

c
matches any non-metacharacter *c*.

\c
matches a character defined by the same escape sequences used in string constants or the literal character *c* if *c* is not an escape sequence.

.
matches any character (including newline).

^
matches the front of a string.

$
matches the back of a string.

[c1c2c3...]
matches any character in the class c1c2c3... . An interval of characters is denoted c1-c2 inside a class [...].

[^c1c2c3...]
matches any character not in the class c1c2c3...

Character classes are also support, these can only be used inside the brackets of a regular expression and define groups of characters in a portable manner (e.g., /[[:digit:]]/). The list that are supported by Wmawk2 is: alnum, alpha, blank, cntrl, digit, graph, lower, print, space, upper, xdigit. These are defined as:

- alnum – alphanumeric characters (typically a-z A-Z 0-9)
- alpha – alphabetic characters (typically a-z A-Z)
- blank – space and tab characters
- cntrl – control characters
- digit – typically 0-9
- graph – characters that are both printable and visible (e.g., a space is not visible)
- lower – lower case alphabetic characters (typically a-z)
- print – printable characters (characters that are not control characters)
- space – typically space, tab, newline, carriage return, formfeed and vertical tab.
- Upper - upper case alphabetic characters (typically A-Z)
- Xdigit – hex digits (typically 0-9 a-f A-F)

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

***r*1*r*2**
matches *r*1 followed immediately by *r*2 (concatenation).

***r*1 | *r*2**
matches *r*1 or *r*2 (alternation).

r*
matches *r* repeated zero or more times.

***r*+**
matches *r* repeated one or more times.

***r*?**
matches *r* zero or once.

(*r*)
matches *r*, 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.

Wmawk2 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 | Wmawk2 { 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 |
Wmawk2 '{ 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. **Wmawk2** also allows **delete** *array*, which deletes all elements of *array*.

Multidimensional arrays are synthesized with concatenation using the built-in variable **SUBSEP**. *array*[*expr*1, *expr*2] is equivalent to *array*[*expr*1 **SUBSEP** *expr*2]. 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.

ARGC
number of command line arguments.

ARGV
array of command line arguments, 0..ARGC-1.

CONVFMT
format for internal conversion of numbers to string, initially = "%.6g".

ENVIRON
array indexed by environment variables. An environment string, *var=value* is stored as **ENVIRON**[*var*] = *value*.

FILENAME
name of the current input file.

FNR
current record number in **FILENAME**.

FS
splits records into fields as a regular expression.

NF
number of fields in the current record.

NR
current record number in the total input stream.

OFMT
format for printing numbers; initially = "%.6g".

OFS
inserted between fields on output, initially = " ".

ORS
terminates each record on output, initially = "\n".

RLENGTH
length set by the last call to the built-in function, **match()**.

RS
input record separator, initially = "\n".

RSTART
index set by the last call to **match()**.

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*.

If s is an array length returns the number of elements in the array.

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 -π and π.
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. Wmawk2 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. If you need a pseudo random number generator that will give the same sequence of random numbers on all awk implementations see prand.awk in AWK-library at https://github.com/p-j-miller/
systime()	Wmawk2 extension. Returns the current time (UTC) as the number of seconds since 1970-01-0100:00:00 UTC (the "Epoch"). This always returns an integer number of seconds. In April 2020 [just over 50 years after 1970-01-0100:00:00 UTC] it returned 1586616274.
systime(expr)	Wmawk2 extension. If expr evaluates to 0 returns the UTC time in seconds since 1970-01-0100:00:00 UTC with a 1us resolution. If expr evaluates to a non-zero number (1 is recommended for future compatibility), returns local time to 1us resolution. Local time is correctly adjusted for the time zone and daylight savings time. The AWK-library at https://github.com/p-j-miller/ contains functions to convert the result of systime() into a human readable string (date/time), and to convert a human readable string into a number of seconds since the Epoch. Systime(0) is especially useful for timing the execution of awk scripts as it provides 1us resolution.

9. Input and output

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

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

print *expr*1, *expr*2, ..., *expr*n
writes *expr*1 **OFS** *expr*2 **OFS** ... *expr*n **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, **Wmawk2** associates the filename "/dev/stderr" with stderr which allows print and printf to be redirected to stderr. **Wmawk2** 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 windows.

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 Windows 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. Before executing the system command all output buffers are flushed.

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**. **Wmawk2** 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>. **Wmawk2** 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 "".

Wmawk2 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 **Wmawk2** breaks the record into individual characters, and, similarly, split(*s,A,*"") places the individual characters of *s* into *A*.

12. Multi-line records

Since **Wmawk2** 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**. **Wmawk2** 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

```
Wmawk2  -f  prog  v=1  A  t=hello  B
```

ARGC = 5 with **ARGV[0]** = "Wmawk2", **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 **ARGC** equals 1, the input stream is set to stdin, else the command line arguments **ARGV[1]... ARGV[ARGC-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, *expr*1, *expr*2 , matches every record between the match of *expr*1 and the match *expr*2 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.

The nextfile statement causes wmawk2 to process stop processing the current file and to start processing the next file specified on the command line. This means that FILENAME is changed, FNR is reset to 1 and processing starts over with the first rule in the program.

14. Unicode Support

Version 1.2 of Wmawk2 adds utf-8 functionality (" Unicode") for the command line, environment variables, filenames and system/pipe commands and allows the console output to display uft8 characters, and utf8 characters to be input from the console via stdin. It does not change any of the string handling or matching functions [including regular expressions] (which still assume 8-bit characters), so for example:

    wmawk2 "BEGIN{print length(\"€\")}"

will print 3 as the '€' (Euro) character takes up 3 bytes when encoded in utf8.

Another change in version 1.2 is that

    wmawk2 "BEGIN {print \"€\"}"

will (correctly) print

    €

Whereas previous versions of wmawk2 would print:

    wmawk2: write error to file /dev/stdout (Invalid argument)

The AWK-library at https://github.com/p-j-miller/ provides awk functions that can be used to determine the width of UTF-8 characters (in bytes), the number of utf8 characters in a string, etc.

In many cases scripts will do what is expected when utf8 Unicode characters are used, but you do need to remember that what appears to be a single utf8 character may take multiple bytes. As a simple example:

    wmawk2 "$1==\"€\" {print $1}"

does work as expected, as does:

    wmawk2 "/^[0-9]+€/ {print }"

but:

```
wmawk2 "/^€+1/ {print }"
```

does not work as expected, for example €€1 will not be matched by the regex and so not printed (as the € uft8 character needs more than 1 byte to represent it). This example is easily fixed as:

```
wmawk2 "/^(€)+1/ {print }"
```

does work as expected.

Note that uft8 files may start with a BOM (or a BOM mark may need adding to a uft8 file to allow it to be recognized correctly by other software) – see https://en.wikipedia.org/wiki/Byte_order_mark for more information.

Note that you cannot use utf8 characters in awk variable, array or function names.

EXAMPLES

1. emulate cat (echo every line)

```
{ print }
```

2. emulate wc (count the number of words and characters in a file)

```
{ 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 – warning this can be very slow for large files.

```
{ 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
}
```

The AWK-library at https://github.com/p-j-miller/ provides awk functions for very fast sorting, median calculation, and conversion between strings and times as seconds since the EPOCH (as returned by systime()). Note that multiple files can be loaded using multiple -f options to wmawk2, so for example:

wmawk2 -f median.awk -f qsort.awk -f median_test.awk

Loads the (library) files median.awk and qsort.awk and then loads and executes the program median_test.awk .

WHY USE AWK/MAWK/WMAWK ?

In general, awk is best at processing largeish text files that can be basically processed a line at a time, it has no graphics support and it's a command line tool so it has no gui.

A good example is given at https://brenocon.com/blog/2009/09/dont-mawk-awk-the-fastest-and-most-elegant-big-data-munging-language/ . This example shows that mawk was both the fastest to execute (vs a range of other languages as well as other awk implementations) and it only tool 3 lines of awk to implement so was probably the fastest to write as well.

If you want more information on awk in general then http://www.awklang.org/ is a good place to start.

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.

Wmawk2 uses its own implementation of printf() and sprint() so the functionality is fixed independent of the functionality in the underlying C library functions. This means that the Null character (\0) is allowed in the format string and argument strings. It also means that %d displays the number as a 64-bit integer. The capabilities are print a character, %c; print a signed integer, %d or %i ; print an unsigned integer, %u, %x, %X or %o; print a floating-point number, %f, %g, %G, %e or %E ; print a string, %s. An optional preceding l is allowed before an integer control character (which is ignored). An optional field width and precision as supported as usual. The porter has tried to ensure the output is identical to that obtained using gawk when run under Windows but that cannot be guaranteed for all versions of gawk and all scripts/datasets. **Wmawk2** also uses its own version of strtod() to convert floating point numbers but this should not impact the underlying functionality (this was done for speed rather than function). It should be noted that **Wmawk2** does not recognize inf, nan or hex string as numbers. This means the data

```
0x4  inf nan
```

Passed to the script

```
{ print 7 + $1, 8 + $2, 9+$3}
```

Will result in

7 8 9

I.e., the 0x4 inf and nan are all treated as 0 when used as numbers. This was done for backwards compatibility with earlier versions of awk.

Note that **Wmawk2** uses double precision floating point numbers (as defined in IEE 754) internally where required. These provide 15 to 17 significant digits (53 binary bits), have a maximum value of +/- 1.797e+308 and a smallest non-zero number of +/-4.94e-324. As there are only 53 binary bits available in the mantissa, while integers are printed by sprintf and printf to 64 bits not all of these may be exact.

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**.

Wmawk2, 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, **Wmawk2** 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 **Wmawk2**, 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, **Wmawk2** 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, **Wmawk2** 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 **Wmawk2** 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.

Note that gawk has a number of extensions that are not present in mawk/Wmawk2. In general, these are easy to work around with awk scripts. As an example, sorting can be done with the functions in the qsort.awk program in the examples directory, the script at https://docstore.mik.ua/orelly/unix3/sedawk/ch09_03.htm or you can use the sort program at https://github.com/p-j-miller/nsort either via pipes or by using the awk system command. The AWK-library at https://github.com/p-j-miller/ provides awk functions for very fast sorting, median calculation, and conversion between strings and times as seconds since the EPOCH (as returned by systime()).

SEE ALSO

Egrep (UNIX/Linux command)

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.

https://dev.to/rrampage/awk---a-useful-little-language-2fhf a useful, but basic introduction to awk.

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.

http://www.awklang.org/ which claims to be the site for things related to the awk language. This includes links to 5 awk books., tutorials and a number of examples including one that can be run live on the web site to deal the cards for poker hands.

https://github.com/e36freak/awk-libs a library of awk functions including a number of different sorts, extra functions for math's, strings, csv files, etc.

BUGS

Wmawk2 cannot handle ascii NUL \0 in source (script) files. It is allowed in data files, and you can output NUL using printf with %c or by printing a string containing a NUL character.

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

INSTALLATION

The executable (for 64-bit windows) and this manual as a pdf are available from github at https://github.com/p-j-miller/wmawk2 . The executable is a portable program and may be placed anywhere (even on removable media). It is suggested that you place the directory where the executable is on your PATH (it's easy to find instructions on how to do this on the web for example search "adding a directory to path windows 10") – if this is not done then you need to include the path to the executable to invoke **Wmawk2** if you are not in the same directory as the executable.

Alternatively, you can compile from source – a dev-c++ project file is provided in the github repository.

AUTHOR

This windows port was created by Peter Miller.

Mawk was written by Mike Brennan.

LICENSE

Wmawk2 is distributed without warranty under the terms of the GNU General Public License, version 3, 2007.

The files atof.*, unicode.* and ya_sprintf are under the MIT license (see sources code).

The source code is available from https://github.com/p-j-miller/wmawk2

VERSIONS

Version **1.0** was the first public release. This was compiled with TDM-GCC 9.2.0, and uses MSVCRT as runtime library, which is available on all versions of Windows (but may need to be downloaded separately from Microsoft).

Version **1.1** was compiled using GCC 14.1.0 x64 (using the UCRT runtime) for Windows. The UCRT is effectively built into the operating system in Windows 10 and 11 and can be installed on earlier versions of Windows (see https://support.microsoft.com/en-gb/topic/update-for-universal-c-runtime-in-windows-c0514201-7fe6-95a3-b0a5-287930f3560c). Some buffer sizes were also optimized for file i/o, and an internal random number generator is now used which is "more random" than the one used previously. There was no change in user functionality with 1v1, but the execution times for test programs may have changed (in some cases becoming faster).

Version **2.0** was compiled using GCC 14.2.0 x64 (using the UCRT runtime) for Windows. This version adds utf-8 functionality (" Unicode") for the command line, environment variables, filenames and system/pipe commands and allows the console output to display uft8 characters, and utf8 characters to be input from the console via stdin. It does not change any of the string handling or matching functions [including regular expressions] (which still assume 8-bit characters). For more details see the "Unicode support" section of this manual. Memory allocation was also changed to improve its speed and, in some conditions, it will use (potentially a lot) less memory, while in other situations it may use a little more memory than previously. Hexadecimal constants (staring 0X or 0X) are now allowed in awk scripts. Systime(0) and systime(1) functions added.