AWK

AWK (*awk*)[4] is a domain-specific language designed for text processing and typically used as a data extraction and reporting tool. Like sed and grep, it's a filter,[4] and is a standard feature of most Unix-like operating systems.

The AWK language is a data-driven scripting language consisting of a set of actions to be taken against streams of textual data – either run directly on files or used as part of a pipeline – for purposes of extracting or transforming text, such as producing formatted reports. The language extensively uses the string datatype, associative arrays (that is, arrays indexed by key strings), and regular expressions. While AWK has a limited intended application domain and was especially designed to support one-liner programs, the language is Turing-complete, and even the early Bell Labs users of AWK often wrote well-structured large AWK programs.[5]

AWK was created at Bell Labs in the 1970s,[6] and its name is derived from the surnames of its authors: Alfred Aho, Peter Weinberger, and Brian Kernighan. The acronym is pronounced the same as the bird auk, which is on the cover of *The AWK Programming Language*.[7] When written in all lowercase letters, as `awk`, it refers to the Unix or Plan 9 program that runs scripts written in the AWK programming language.

Contents

AWK

Paradigm	Scripting, procedural, data-driven[1]
Designed by	Alfred Aho, Peter Weinberger, and Brian Kernighan
First appeared	1977
Stable release	IEEE Std 1003.1-2008 (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html) (POSIX) / 1985
Typing discipline	none; can handle strings, integers and floating-point numbers; regular expressions
OS	Cross-platform
Major implementations	
awk, GNU Awk, mawk, nawk, MKS AWK, Thompson AWK (compiler), Awka (compiler)	
Dialects	
old awk oawk 1977, *new awk* nawk 1985, *GNU Awk* gawk	
Influenced by	
C, sed, SNOBOL[2][3]	
Influenced	
Tcl, AMPL, Perl, Korn Shell (*ksh93*, *dtksh*, *tksh*), Lua	

History

AWK was initially developed in 1977 by Alfred Aho (author of egrep), Peter J. Weinberger (who worked on tiny relational databases), and Brian Kernighan; it takes its name from their respective initials. According to Kernighan, one of the goals of AWK was to have a tool that would easily manipulate both numbers and strings. AWK was also inspired by Marc Rochkind's programming language that was used to search for patterns in input data, and was implemented using yacc.[8]

As one of the early tools to appear in Version 7 Unix, AWK added computational features to a Unix pipeline besides the Bourne shell, the only scripting language available in a standard Unix environment. It is one of the mandatory utilities of the Single UNIX Specification,[9] and is required by the Linux Standard Base specification.[10]

AWK was significantly revised and expanded in 1985–88, resulting in the GNU AWK implementation written by Paul Rubin, Jay Fenlason, and Richard Stallman, released in 1988.[11] GNU AWK may be the most widely deployed version[12] because it is included with GNU-based Linux packages. GNU AWK has been maintained solely by Arnold Robbins since 1994.[11] Brian Kernighan's nawk (New AWK) source was first released in 1993 unpublicized, and publicly since the late 1990s; many BSD systems use it to avoid the GPL license.[11]

AWK was preceded by sed (1974). Both were designed for text processing. They share the line-oriented, data-driven paradigm, and are particularly suited to writing one-liner programs, due to the implicit main loop and current line variables. The power and terseness of early AWK programs – notably the powerful regular expression handling and conciseness due to implicit variables, which facilitate one-liners – together with the limitations of AWK at the time, were important inspirations for the Perl language (1987). In the 1990s, Perl became very popular, competing with AWK in the niche of Unix text-processing languages.

Structure of AWK programs

> AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed.
>
> — Alfred V. Aho[13]

An AWK program is a series of pattern action pairs, written as:

```
condition { action }
condition { action }
...
```

where *condition* is typically an expression and *action* is a series of commands. The input is split into records, where by default records are separated by newline characters so that the input is split into lines. The program tests each record against each of the conditions in turn, and executes the *action* for each expression that is true. Either the condition or the action may be omitted. The condition defaults to matching every record. The default action is to print the record. This is the same pattern-action structure as sed.

In addition to a simple AWK expression, such as `foo == 1` or `/^foo/`, the condition can be `BEGIN` or `END` causing the action to be executed before or after all records have been read, or *pattern1*, *pattern2* which matches the range of records starting with a record that matches *pattern1* up to and including the record that matches *pattern2* before again trying to match against *pattern1* on future lines.

In addition to normal arithmetic and logical operators, AWK expressions include the tilde operator, `~`, which matches a regular expression against a string. As handy syntactic sugar, */regexp/* without using the tilde operator matches against the current record; this syntax derives from sed, which in turn inherited it from the ed editor, where `/` is used for searching. This syntax of using slashes as delimiters for regular expressions was subsequently adopted by Perl and ECMAScript, and is now common. The tilde operator was also adopted by Perl.

Commands

AWK commands are the statements that are substituted for *action* in the examples above. AWK commands can include function calls, variable assignments, calculations, or any combination thereof. AWK contains built-in support for many functions; many more are provided by the various flavors of AWK. Also, some flavors support the inclusion of dynamically linked libraries, which can also provide more functions.

The *print* command

The *print* command is used to output text. The output text is always terminated with a predefined string called the output record separator (ORS) whose default value is a newline. The simplest form of this command is:

`print`
: This displays the contents of the current record. In AWK, records are broken down into *fields*, and these can be displayed separately:

`print $1`
: Displays the first field of the current record

`print $1, $3`
: Displays the first and third fields of the current record, separated by a predefined string called the output field separator (OFS) whose default value is a single space character

Although these fields (*$X*) may bear resemblance to variables (the $ symbol indicates variables in Perl), they actually refer to the fields of the current record. A special case, *$0*, refers to the entire record. In fact, the commands "`print`" and "`print $0`" are identical in functionality.

The *print* command can also display the results of calculations and/or function calls:

```
/regex_pattern/ {
    # Actions to perform in the event the record (line) matches the above regex_pattern
    print 3+2
    print foobar(3)
    print foobar(variable)
    print sin(3-2)
}
```

Output may be sent to a file:

```
/regex_pattern/ {
    # Actions to perform in the event the record (line) matches the above regex_pattern
    print "expression" > "file name"
}
```

or through a pipe:

```
/regex_pattern/ {
    # Actions to perform in the event the record (line) matches the above regex_pattern
    print "expression" | "command"
}
```

Built-in variables

Awk's built-in variables include the field variables: $1, $2, $3, and so on ($0 represents the entire record). They hold the text or values in the individual text-fields in a record.

Other variables include:

- `NR`: 'N'umber of 'R'ecords: keeps a current count of the number of input records read so far from all data files. It starts at zero, but is never automatically reset to zero.[14]
- `FNR`: 'F'ile 'N'umber of 'R'ecords: keeps a current count of the number of input records read so far *in the current file.* This variable is automatically reset to zero each time a new file is started.[14]
- `NF`: 'N'umber of 'F'ields: contains the number of fields in the current input record. The last field in the input record can be designated by $NF, the 2nd-to-last field by $(NF-1), the 3rd-to-last field by $(NF-2), etc.
- `FILENAME`: Contains the name of the current input-file.
- `FS`: 'F'ield 'S'eparator: contains the "field separator" character used to divide fields in the input record. The default, "white space", includes any space and tab characters. FS can be reassigned to another character to change the field separator.
- `RS`: 'R'ecord 'S'eparator: stores the current "record separator" character. Since, by default, an input line is the input record, the default record separator character is a "newline".
- `OFS`: 'O'utput 'F'ield 'S'eparator: stores the "output field separator", which separates the fields when Awk prints them. The default is a "space" character.
- `ORS`: 'O'utput 'R'ecord 'S'eparator: stores the "output record separator", which separates the output records when Awk prints them. The default is a "newline" character.
- `OFMT`: 'O'utput 'F'or'M'a'T': stores the format for numeric output. The default format is "%.6g".

Variables and syntax

Variable names can use any of the characters [A-Za-z0-9_], with the exception of language keywords. The operators + - * / represent addition, subtraction, multiplication, and division, respectively. For string concatenation, simply place two variables (or string constants) next to each other. It is optional to use a space in between if string constants are involved, but two variable names placed adjacent to each other require a space in between. Double quotes delimit string constants. Statements need not end with semicolons. Finally, comments can be added to programs by using # as the first character on a line.

User-defined functions

In a format similar to C, function definitions consist of the keyword `function`, the function name, argument names and the function body. Here is an example of a function.

```
function add_three (number) {
    return number + 3
}
```

This statement can be invoked as follows:

```
(pattern)
{
   print add_three(36)     # Outputs '''39'''
}
```

Functions can have variables that are in the local scope. The names of these are added to the end of the argument list, though values for these should be omitted when calling the function. It is convention to add some whitespace in the argument list before the local variables, to indicate where the parameters end and the local variables begin.

Examples

Hello World

Here is the customary "Hello, world" program written in AWK:

```
BEGIN { print "Hello, world!" }
```

Note that an explicit `exit` statement is not needed here; since the only pattern is `BEGIN`, no command-line arguments are processed.

Print lines longer than 80 characters

Print all lines longer than 80 characters. Note that the default action is to print the current line.

```
length($0) > 80
```

Count words

Count words in the input and print the number of lines, words, and characters (like wc):

```
{
    words += NF
    chars += length + 1 # add one to account for the newline character at the end of each record
(line)
}
END { print NR, words, chars }
```

As there is no pattern for the first line of the program, every line of input matches by default, so the increment actions are executed for every line. Note that `words += NF` is shorthand for `words = words + NF`.

Sum last word

```
{ s += $NF }
END { print s + 0 }
```

s is incremented by the numeric value of *$NF*, which is the last word on the line as defined by AWK's field separator (by default, white-space). *NF* is the number of fields in the current line, e.g. 4. Since *$4* is the value of the fourth field, *$NF* is the value of the last field in the line regardless of how many fields this line has, or whether it has more or fewer fields than surrounding lines. $ is actually a unary operator with the highest operator precedence. (If the line has no fields, then *NF* is 0, *$0* is the whole line, which in this case is empty apart from possible white-space, and so has the numeric value 0.)

At the end of the input the *END* pattern matches, so *s* is printed. However, since there may have been no lines of input at all, in which case no value has ever been assigned to *s*, it will by default be an empty string. Adding zero to a variable is an AWK idiom for coercing it from a string to a numeric value. (Concatenating an empty string is to coerce from a number to a string, e.g. *s ""*. Note, there's no operator to concatenate strings, they're just placed adjacently.) With the coercion the program prints "0" on an empty input, without it an empty line is printed.

Match a range of input lines

```
NR % 4 == 1, NR % 4 == 3 { printf "%6d  %s\n", NR, $0 }
```

The action statement prints each line numbered. The printf function emulates the standard C printf and works similarly to the print command described above. The pattern to match, however, works as follows: *NR* is the number of records, typically lines of input, AWK has so far read, i.e. the current line number, starting at 1 for the first line of input. *%* is the modulo operator. *NR % 4 == 1* is true for the 1st, 5th, 9th, etc., lines of input. Likewise, *NR % 4 == 3* is true for the 3rd, 7th, 11th, etc., lines of input. The range pattern is false until the first part matches, on line 1, and then remains true up to and including when the second part matches, on line 3. It then stays false until the first part matches again on line 5.

Thus, the program prints lines 1,2,3, skips line 4, and then 5,6,7, and so on. For each line, it prints the line number (on a 6 character-wide field) and then the line contents. For example, when executed on this input:

```
Rome
Florence
Milan
Naples
Turin
Venice
```

The previous program prints:

```
     1 Rome
     2 Florence
     3 Milan
     5 Turin
     6 Venice
```

Printing the initial or the final part of a file

As a special case, when the first part of a range pattern is constantly true, e.g. *1*, the range will start at the beginning of the input. Similarly, if the second part is constantly false, e.g. *0*, the range will continue until the end of input. For example,

```
/^--cut here--$/, 0
```

prints lines of input from the first line matching the regular expression ^*--cut here--$*, that is, a line containing only the phrase "--cut here--", to the end.

Calculate word frequencies

Word frequency using associative arrays:

```
BEGIN {
    FS="[^a-zA-Z]+"
}
{
    for (i=1; i<=NF; i++)
        words[tolower($i)]++
}
END {
    for (i in words)
        print i, words[i]
}
```

The BEGIN block sets the field separator to any sequence of non-alphabetic characters. Note that separators can be regular expressions. After that, we get to a bare action, which performs the action on every input line. In this case, for every field on the line, we add one to the number of times that word, first converted to lowercase, appears. Finally, in the END block, we print the words with their frequencies. The line

```
for (i in words)
```

creates a loop that goes through the array *words*, setting *i* to each *subscript* of the array. This is different from most languages, where such a loop goes through each *value* in the array. The loop thus prints out each word followed by its frequency count. `tolower` was an addition to the One True awk (see below) made after the book was published.

Match pattern from command line

This program can be represented in several ways. The first one uses the Bourne shell to make a shell script that does everything. It is the shortest of these methods:

```
#!/bin/sh

pattern="$1"
shift
awk '/'"$pattern"'/ { print FILENAME ":" $0 }' "$@"
```

The `$pattern` in the awk command is not protected by single quotes so that the shell does expand the variable but it needs to be put in double quotes to properly handle patterns containing spaces. A pattern by itself in the usual way checks to see if the whole line (`$0`) matches. `FILENAME` contains the current filename.

awk has no explicit concatenation operator; two adjacent strings concatenate them. `$0` expands to the original unchanged input line.

There are alternate ways of writing this. This shell script accesses the environment directly from within awk:

```
#!/bin/sh

export pattern="$1"
shift
awk '$0 ~ ENVIRON["pattern"] { print FILENAME ":" $0 }' "$@"
```

This is a shell script that uses `ENVIRON`, an array introduced in a newer version of the One True awk after the book was published. The subscript of `ENVIRON` is the name of an environment variable; its result is the variable's value. This is like the getenv function in various standard libraries and POSIX. The shell script makes an environment variable `pattern` containing the first argument, then drops that argument and has awk look for the pattern in each file.

`~` checks to see if its left operand matches its right operand; `!~` is its inverse. Note that a regular expression is just a string and can be stored in variables.

The next way uses command-line variable assignment, in which an argument to awk can be seen as an assignment to a variable:

```
#!/bin/sh

pattern="$1"
shift
awk '$0 ~ pattern { print FILENAME ":" $0 }' "pattern=$pattern" "$@"
```

Or You can use the *-v var=value* command line option (e.g. *awk -v pattern="$pattern" ...*).

Finally, this is written in pure awk, without help from a shell or without the need to know too much about the implementation of the awk script (as the variable assignment on command line one does), but is a bit lengthy:

```
BEGIN {
    pattern = ARGV[1]
    for (i = 1; i < ARGC; i++) # remove first argument
        ARGV[i] = ARGV[i + 1]
    ARGC--
    if (ARGC == 1) { # the pattern was the only thing, so force read from standard input (used
by book)
        ARGC = 2
        ARGV[1] = "-"
    }
}
$0 ~ pattern { print FILENAME ":" $0 }
```

The `BEGIN` is necessary not only to extract the first argument, but also to prevent it from being interpreted as a filename after the `BEGIN` block ends. `ARGC`, the number of arguments, is always guaranteed to be ≥1, as `ARGV[0]` is the name of the command that executed the script, most often the string `"awk"`. Also note that `ARGV[ARGC]` is the empty string, `""`. # initiates a comment that expands to the end of the line.

Note the `if` block. awk only checks to see if it should read from standard input before it runs the command. This means that

```
awk 'prog'
```

only works because the fact that there are no filenames is only checked before `prog` is run! If you explicitly set `ARGC` to 1 so that there are no arguments, awk will simply quit because it feels there are no more input files. Therefore, you need to explicitly say to read from standard input with the special filename `-`.

Self-contained AWK scripts

On Unix-like operating systems self-contained AWK scripts can be constructed using the shebang syntax.

For example, a script that prints the content of a given file may be built by creating a file named `print.awk` with the following content:

```
#!/usr/bin/awk -f
{ print $0 }
```

It can be invoked with: `./print.awk <filename>`

The `-f` tells AWK that the argument that follows is the file to read the AWK program from, which is the same flag that is used in sed. Since they are often used for one-liners, both these programs default to executing a program given as a command-line argument, rather than a separate file.

Versions and implementations

AWK was originally written in 1977 and distributed with Version 7 Unix.

In 1985 its authors started expanding the language, most significantly by adding user-defined functions. The language is described in the book *The AWK Programming Language*, published 1988, and its implementation was made available in releases of UNIX System V. To avoid confusion with the incompatible older version, this version was sometimes called "new awk" or *nawk*. This implementation was released under a free software license in 1996 and is still maintained by Brian Kernighan (see external links below).

Old versions of Unix, such as UNIX/32V, included `awkcc`, which converted AWK to C. Kernighan wrote a program to turn awk into C++; its state is not known.[15]

- **BWK awk**, also known as **nawk**, refers to the version by Brian Kernighan. It has been dubbed the "One True AWK" because of the use of the term in association with the book that originally described the language and the fact that Kernighan was one of the original authors of AWK.[7] FreeBSD refers to this version as *one-true-awk*.[16] This version also has features not in the book, such as `tolower` and `ENVIRON` that are explained above; see the FIXES file in the source archive for details. This version is used by, for example, Android, FreeBSD, NetBSD, OpenBSD, macOS, and illumos. Brian Kernighan and Arnold Robbins are the main contributors to a source repository for *nawk*: github.com/onetrueawk/awk (https://github.com/onetrueawk/awk).
- **gawk** (GNU awk) is another free-software implementation and the only implementation that makes serious progress implementing internationalization and localization and TCP/IP networking. It was written before the original implementation became freely available. It includes its own debugger, and its profiler enables the user to make measured performance enhancements to a script. It also enables the user to extend functionality with shared libraries. Some Linux distributions include *gawk* as their default AWK implementation.
  - **gawk-csv**. The CSV extension of *gawk* provides facilities for handling input and output CSV formatted data.[17]

- **mawk** is a very fast AWK implementation by Mike Brennan based on a bytecode interpreter.
- **libmawk** is a fork of mawk, allowing applications to embed multiple parallel instances of awk interpreters.
- **awka** (whose front end is written atop the *mawk* program) is another translator of AWK scripts into C code. When compiled, statically including the author's libawka.a, the resulting executables are considerably sped up and, according to the author's tests, compare very well with other versions of AWK, Perl, or Tcl. Small scripts will turn into programs of 160–170 kB.
- **tawk** (Thompson AWK) is an AWK compiler for Solaris, DOS, OS/2, and Windows, previously sold by Thompson Automation Software (which has ceased its activities).[18]
- **Jawk** is a project to implement AWK in Java, hosted on SourceForge.[19] Extensions to the language are added to provide access to Java features within AWK scripts (i.e., Java threads, sockets, collections, etc.).
- **xgawk** is a fork of *gawk*[20] that extends *gawk* with dynamically loadable libraries. The XMLgawk extension was integrated into the official GNU Awk release 4.1.0.
- **QSEAWK** is an embedded AWK interpreter implementation included in the QSE library that provides embedding application programming interface (API) for C and C++.[21]
- **libfawk** is a very small, function-only, reentrant, embeddable interpreter written in C
- **BusyBox** includes an AWK implementation written by Dmitry Zakharov. This is a very small implementation suitable for embedded systems.

Books

- Aho, Alfred V.; Kernighan, Brian W.; Weinberger, Peter J. (1988-01-01). *The AWK Programming Language* (https://archive.org/details/awkprogrammingla00ahoa). New York, NY: Addison-Wesley. ISBN 0-201-07981-X. Retrieved 2017-01-22.
- Robbins, Arnold (2001-05-15). *Effective awk Programming* (http://www.oreilly.com/catalog/awkprog3/) (3rd ed.). Sebastopol, CA: O'Reilly Media. ISBN 0-596-00070-7. Retrieved 2009-04-16.
- Dougherty, Dale; Robbins, Arnold (1997-03-01). *sed & awk* (http://www.oreilly.com/catalog/sed2/) (2nd ed.). Sebastopol, CA: O'Reilly Media. ISBN 1-56592-225-5. Retrieved 2009-04-16.
- Robbins, Arnold (2000). *Effective Awk Programming: A User's Guide for Gnu Awk* (https://www.gnu.org/software/gawk/manual/) (1.0.3 ed.). Bloomington, IN: iUniverse. ISBN 0-595-10034-1. Archived (https://web.archive.org/web/20090412190359/https://www.gnu.org/software/gawk/manual/) from the original on 12 April 2009. Retrieved 2009-04-16.

See also

- Data transformation
- Event-driven programming
- List of Unix commands
- sed

References

1. Stutz, Michael (September 19, 2006). "Get started with GAWK: AWK language fundamentals" (https://www6.software.ibm.com/developerworks/education/au-gawk/au-gawk-a4.pdf) (PDF). *developerWorks*. IBM. Retrieved 2015-01-29. "[AWK is] often called a data-driven language -- the program statements describe the input data to match and process rather than a sequence of program steps"

2. Andreas J. Pilavakis (1989). *UNIX Workshop*. Macmillan International Higher Education. p. 196.
3. Arnold Robbins (2015). *Effective Awk Programming: Universal Text Processing and Pattern Matching* (4th ed.). O'Reilly Media. p. 560.
4. James W. Livingston (May 2, 1988). "The Great awk Program is No Birdbrain". *Digital Review*. p. 91.
5. Raymond, Eric S. "Applying Minilanguages" (https://web.archive.org/web/20080730063308/http://www.faqs.org/docs/artu/ch08s02.html#awk). *The Art of Unix Programming*. Case Study: awk. Archived from the original (http://www.faqs.org/docs/artu/ch08s02.html#awk) on July 30, 2008. Retrieved May 11, 2010. "The awk action language is Turing-complete, and can read and write files."
6. Aho, Alfred V.; Kernighan, Brian W.; Weinberger, Peter J. (September 1, 1978). *Awk — A Pattern Scanning and Processing Language (Second Edition)* (https://wolfram.schneider.org/bsd/7thEdManVol2/awk/awk.html) (Technical report). Unix Seventh Edition Manual, Volume 2. Bell Telephone Laboratories, Inc.
7. Aho, Alfred V.; Kernighan, Brian W.; Weinberger, Peter J. (1988). *The AWK Programming Language* (https://archive.org/details/pdfy-MgN0H1joIoDVoIC7). Addison-Wesley Publishing Company. ISBN 9780201079814. Retrieved 16 May 2015.
8. "UNIX Special: Profs Kernighan & Brailsford" (https://www.youtube.com/watch?v=vT_J6xc-Az0). *Computerphile*. September 30, 2015.
9. "The Single UNIX Specification, Version 3, Utilities Interface Table" (https://web.archive.org/web/20180105030249/http://www.unix.org/version3/apis/cu.html). Archived from the original (http://www.unix.org/version3/apis/cu.html) on 2018-01-05. Retrieved 2005-12-18.
10. "Chapter 15. Commands and Utilities". *Linux Standard Base Core Specification 4.0* (https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic.html#COMMAND) (Technical report). Linux Foundation. 2008.
11. Robbins, Arnold (March 2014). "The GNU Project and Me: 27 Years with GNU AWK" (http://www.skeeve.com/gnu-awk-and-me-2014.pdf) (PDF). *skeeve.com*. Retrieved October 4, 2014.
12. Dougherty, Dale; Robbins, Arnold (1997). *sed & awk* (2nd ed.). Sebastopol, CA: O'Reilly. p. 221. ISBN 1-565-92225-5.
13. Hamilton, Naomi (May 30, 2008). "The A-Z of Programming Languages: AWK" (https://www.computerworld.com/article/2535126/the-a-z-of-programming-languages--awk.html). *Computerworld*. Retrieved 2008-12-12.
14. https://www.gnu.org/software/gawk/manual/html_node/Records.html#index-FNR-variable
15. Kernighan, Brian W. (April 24–25, 1991). *An AWK to C++ Translator* (https://www.cs.princeton.edu/~bwk/btl.mirror/awkc++.pdf) (PDF). Usenix C++ Conference. Washington, DC. pp. 217–228.
16. "FreeBSD's work log for importing BWK awk into FreeBSD's core" (http://www.freebsd.org/cgi/cvsweb.cgi/src/contrib/one-true-awk/FREEBSD-upgrade?rev=1.9&content-type=text/x-cvsweb-markup). May 16, 2005. Archived (https://web.archive.org/web/20130908180035/http://www.freebsd.org/cgi/cvsweb.cgi/src/contrib/one-true-awk/FREEBSD-upgrade?rev=1.9&content-type=text%2Fx-cvsweb-markup) from the original on September 8, 2013. Retrieved September 20, 2006.
17. **gawk-csv** documentation at http://gawkextlib.sourceforge.net/csv/gawk-csv.html
18. James K. Lawless (May 1, 1997). "Examining the TAWK Compiler" (https://www.drdobbs.com/tools/examining-the-tawk-compiler/184410193). *Dr. Dobb's Journal*.
19. *Jawk* at SourceForge (http://sourceforge.net/projects/jawk/)
20. *xgawk* Home Page (http://gawkextlib.sourceforge.net/)
21. QSEAWK at GitHub (https://github.com/hyung-hwan/qse)

Further reading

- Hamilton, Naomi (May 30, 2008). "The A-Z of Programming Languages: AWK" (https://www.computerworld.com/article/2535126/the-a-z-of-programming-languages--awk.html). *Computerworld*. Retrieved 2008-12-12. – Interview with Alfred V. Aho on AWK
- Robbins, Daniel (2000-12-01). "Awk by example, Part 1: An intro to the great language with the strange name" (http://www.ibm.com/developerworks/library/l-awk1/). *Common threads*. IBM DeveloperWorks. Retrieved 2009-04-16.
- Robbins, Daniel (2001-01-01). "Awk by example, Part 2: Records, loops, and arrays" (http://www.ibm.com/developerworks/library/l-awk2/). *Common threads*. IBM DeveloperWorks. Retrieved 2009-04-16.
- Robbins, Daniel (2001-04-01). "Awk by example, Part 3: String functions and ... checkbooks?" (http://www.ibm.com/developerworks/library/l-awk3/). *Common threads*. IBM DeveloperWorks. Archived (https://web.archive.org/web/20090519074032/http://www.ibm.com/developerworks/linux/library/l-awk3.html) from the original on 19 May 2009. Retrieved 2009-04-16.
- AWK – Become an expert in 60 minutes (https://web.archive.org/web/20081031084509/http://www.think-lamp.com/2008/10/awk-a-boon-for-cli-enthusiasts/)
- `awk (https://www.opengroup.org/onlinepubs/9699919799/utilities/awk.html)`: pattern scanning and processing language – Commands & Utilities Reference, The Single UNIX Specification, Issue 7 from The Open Group
- `gawk(1) (http://man7.org/linux/man-pages/man1/gawk.1.html)` – Linux User's Manual – User Commands
- Gawkinet (https://www.gnu.org/software/gawk/manual/gawkinet/): TCP/IP Internetworking with Gawk

External links

- The Amazing Awk Assembler (https://web.archive.org/web/20120207095326/http://doc.cat-v.org/henry_spencer/amazing_awk_assembler/) by Henry Spencer.
- AWK (https://curlie.org/Computers/Programming/Languages/Awk) at Curlie
- awklang.org (http://awklang.org) The site for things related to the awk language

---