AWK

A Text Processing Language

Nelson H.F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, UT 84112
USA
Tel: +1 801 581 5254
FAX: +1 801 581 4148
E-mail: <beebe@math.utah.edu>

Overview

- What is **awk**?
- Background
- Documentation on **awk**
- Related Programs and Languages
- **awk** syntax summary
- Built-in arithmetic functions

Overview ...

- Built-in string functions
- Built-in variables
- Simple **awk** programs
- Case studies

What is awk?

- **awk** is a language that provides easy and powerful facilities for string processing, with regular-expression pattern matching for selection, and subdivision of input records into counted *fields*.
- The name **awk** comes from the initials of its authors, A. V. **A**ho, P. J. **W**einberger, and B. W. **K**ernighan.
- After **make**, **awk** is probably the next most useful tool on the UNIX workbench.

Background

1978 A. V. Aho, P. J. Weinberger, and B. W. Kernighan produce first **awk**.

1978– **awk** distributed as standard part of UNIX, and rapidly becomes a widely-used programming language.

1987 Original authors redesign **awk**, and write a book about the new language, **nawk**, an upward-compatible superset of the old language. Source code for the new language is made available under license from AT&T.

Background ...

1987 **nawk** ported to TOPS-20, VAX VMS, and PC DOS by NHFB, and the portable version returned to AT&T with many comments and recommendations.

1988 Improved **nawk** appears incorporating several of NHFB's ideas.

1988 Free Software Foundation develops first prototype of independent implementation of **nawk** subset, to be called **gawk**. **gawk** runs on UNIX and PC DOS.

1989 Commercial implementations for PC DOS appear from Mortice Kern Systems and Polytron.

Background ...

1989 **nawk** becomes standard part of AT&T System VR4 UNIX, and part of IEEE POSIX.

1989 **nawk** regular expressions extended for international use in X/Open specifications.

1989 FSF's **gawk** reaches compatibility with complete **nawk** language, and its authors collaborate with AT&T in plans for extending and improving the language.

1990 **gawk** ported to VAX VMS by NHFB.

Documentation on awk

- A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, 1988.
- D. B. Close, A. D. Robbins, P. H. Rubin, and R. M. Stallman, *The GAWK Manual*, Free Software Foundation, 1989.

Related Programs and Languages

snobol Pattern-matching language developed in late 1960s.

snobol4 R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, 1971.

spitbol Descendant of **snobol**, developed in 1970s.

C Main implementation language of UNIX, designed in 1971.

sed UNIX stream editor, 1978.

Related Programs and Languages ...

grep UNIX pattern-matching filter (from **ed** editor command, **g/re/p**, global regular-expression print), 1978.

icon Distant descendant of **snobol**, but with block-structured language syntax. Described in R. E. Griswold and Madge T. Griswold, *The Icon Programming Language*, Prentice-Hall, 1983, and *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.

perl Larry Wall's **p**attern **e**xtraction **r**eport **l**anguage, 1988. Book: L. Wall and R. Schwarz, *Programming perl*, O'Reilly, 1991.

awk Syntax Summary

- Case-sensitive, like C.
- Control structures from C language.
  - { statement; ...; statement; }
  - **if** (expression) statement
  - **if** (expression) statement1 **else** statement2
  - **while** (expression) statement
  - **for** (expr1**;** expr2**;** expr3) statement
  - **for** (variable **in** array) statement
  - **do** statement **while** (expression)

awk Syntax Summary ...

- Control structures
  - **break**
  - **continue**
  - **next**
  - **exit**
  - **exit** expression
- Statements must normally fit on one line, except that breaks after a comma, or with a backslash-newline, are permitted.
- Comments go from **#** to end of line.

awk Syntax Summary ...

- Operators include

`+ - * /`	usual arithmetic
`^`	exponentiation (NB: differs from C)
`%`	remainder
`++`	increment
`--`	decrement
`&&`	AND
`||`	OR
`!`	NOT
`~`	regexp match

awk Syntax Summary ...

- Operators ...

`!~`	regexp no match
`%= *= += -= /= ^=`	assignment shorthands, as in C
`> >= < <= ==`	comparisons
`? :`	conditional

awk Syntax Summary ...

```
Fortran                 Awk
k = k + 1               ++k or ++k

n = 5*n                 n *= 5

(x .eq. y)              (x == y)

(a .and. b)             (a && b)

mod(m,n)                m % n

do 10 k = 1,n           for (k = 1; k <= n; ++k)

if (x .gt. 3) then
    y = 4
else                    y = (x > 3) ? 4 : 5
    y = 5
endif
```


awk Syntax Summary ...

- Data types include only floating-point and string scalars, and one-dimensional arrays, **but** array subscripts can be any scalar type (tables in **icon**, or associative memory).

```
table[3.14159] = "table of pi"
age["John"] = 23
birthday["Mary"] = "1974.11.31"
name [1/3] = "one-third"
office["Kelly"] = "123 JWB"
```

- No type declarations; variables are given values by assignment, with type according to context.

awk Syntax Summary ...

- Type coercion: add 0 to a variable to convert to number, concatenate the null string to convert it to a string:

```
n = v + 0
s = v ""
```

- All variables pre-initialized to empty string. This is equivalent to 0 when used as a number.

- No limit on string sizes (thanks to NHFB's suggestions), and all 256 character values, including NULL, may appear in a string.

awk Syntax Summary ...

- Adjacent string expressions are concatenated:

```
string = "ONE" "-AND-" "TWO";
string = "ONE-AND-TWO";
```

are the same.

- Input parsing handled by **awk** language; each input record is available as `$0`, with fields `$1`, `$2`, ..., `$NF`. Out of range field numbers are null strings.

- Fields are normally delimited by white space, but the field separator can be set on the command line, or changed within the program.

awk Syntax Summary ...

- **awk** programs may be given on the command line, or in a specified file.

- Input comes from all files listed on command line, or from standard input. In **nawk**, it may also come from files opened explicitly by the program.

- Output goes to standard output. In **nawk**, it may also be redirected to specified files.

- **system**() function permits running arbitrary programs from inside **awk** program.

awk Syntax Summary ...

- **nawk** introduced functions:

function name(*arg1,...,argn, lcl1,...,lclm*)
{
    statements
}

Local variables declared as extra arguments; otherwise, all **awk** variables are **global**.

No space permitted between name and parenthesized argument list.

One or more spaces conventionally separate actual arguments from local variables in **function** statement.

Built-in Arithmetic Functions

atan2(y,x**)** arctangent of y/x in range $-\pi \ldots +\pi$

cos(x) cosine of x, x in radians

exp(x**)** exponential, e^x

int(x**)** integer part of x (truncates)

log(x**)** base e logarithm of x

rand() random number r, $0 \le r < 1$

sin(x**)** sine of x, x in radians

Built-in Arithmetic Functions ...

sqrt(x**)** square root of x

srand(x**)** supply new seed, x, for **rand**()

Built-in String Functions

gsub(*r*,*s***)** substitute *s* for *r* globally in `$0`, return number of substitutions made

gsub(*r*,*s*,*t***)** substitute *s* for *r* globally in string *t*, return number of substitutions made

index(*s*,*t***)** return first position of string *t* in *s*, or 0 it *t* is not present

length(*s***)** return number of characters in *s*

match(*s*,*r***)** test whether *s* contains a substring matched by *r*; return index or 0; sets `RSTART` and `RLENGTH`

Built-in String Functions

split(*s*,*a***)** split *s* into array *a* on `FS`, return number of fields

split(*s*,*a*,*fs***)** split *s* into array *a* on field separator *fs*, return number of fields

sprintf(*fmt*,*expr-list***)** return *expr-list* formatted according to format string *fmt*

sub(*r*,*s***)** substitute *s* for the leftmost longest substring of `$0` matched by *r*, return number of substitutions made

Built-in String Functions

sub(*r*,*s*,*t***)** substitute *s* for the leftmost longest substring of *t* matched by *r*, return number of substitutions made

substr(*s*,*p***)**
return suffix of *s* starting at position *p* (counting from 1)

substr(*s*,*p*,*n***)**
return suffix of *s* of length *n* starting at position *p* (counting from 1)

Built-in Variables

`ARGC` number of command-line arguments

`ARGV` array of command-line arguments

`FILENAME` name of current input file

`FNR` record number in current input file

`FS` controls the input field separator (default: `" "`)

`NF` number of fields in current record

`NR` number of records read so far

Built-in Variables ...

`OFMT` output format for numbers (default: `"%.6g"`)

`OFS` output field separator (default: `" "`)

`ORS` output record separator (default: `"\n"`)

`RLENGTH` length of string matched by **match** function

`RS` controls the input record separator (default: `"\n"`)

`RSTART` start of string matched by **match** function

`SUBSEP` subscript separator (default: `"\034"`)

Simple awk Programs

- Simple programs look like

  ```
  /optional-regexp/ { statements }
  ```

  or

  ```
  expression { statements }
  ```

  The braced statements are executed for each input line that matches the expression. If the expression is omitted, then the statements are executed for all input lines. If the statements are omitted, matching lines are printed.

Simple awk Programs

```
# print second and seventh fields of input
awk '{print $2, $7;}'

# print long lines
awk 'length($0) > 72'

# print line count
awk 'END { print NR }'

# print sum of column 3 and average
awk '{ sum += $3; }
END { print sum, sum/NR }'

# print sort list of user names
awk -F: '{ print $1 | "sort" }' /etc/passwd
ypcat passwd | \
    awk -F: '{ print $1 | "sort" }'
```


Simple awk Programs

```
# print duplicate words
{
    for (k = 1; k <= NF; ++k)
    {
        frequency[$k]++;
        if (frequency[$k] > 1) print $k;
    }
}

# print words used only once
{
    for (k = 1; k <= NF; ++k)
        frequency[$k]++
}
END {
     for (word in frequency)
       if (frequency[word] == 1) print word;
    }
```


Simple awk Programs ...

- Special patterns BEGIN and END can be used to get control before and after input is read.
- Compound selection patterns combine regular expressions with && (AND), || (OR), ! (NOT), and parentheses.
- Range expressions `/regexp1/,/regexp2/` match all statements between the first line matched by `/regexp1/` and the next line matched by `/regexp2/`, inclusive.

Simple awk Programs ...

- Tilde operator available for more control over pattern matching:

  ```
  expression ~ /regexp/
  expression !~ /regexp/
  ```

  The first is true if `/regexp/` matches a substring of `expression`; the second is true if there is no match.

Case Studies

- 'one-liners' (**awk** book, pp. 17–18)
- table generation
- form letters in LaTeX
- book indexing
- BibTeX bibliography subset extraction
- cross-reference of LaTeX style file macros

Case Studies ...

- indent LaTeX `\begin{} ... \end{}` groups and check for nesting errors
- check `#if ... #endif` nesting in C code
- comment `#if ... #endif` blocks
- expand `#include "..."` directives
- extract complete Fortran `FORMAT` statements
- extract complete C `printf()` statements

Simple awk Programs ...

- Tilde operator available for more control over pattern matching:

  ```
  expression ~ /regexp/
  expression !~ /regexp/
  ```

  The first is true if `/regexp/` matches a substring of `expression`; the second is true if there is no match.