Awk — A Pattern Scanning and Processing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Awk is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *awk* program

length > 72

prints all input lines whose length exceeds 72 characters; the program

NF % 2 == 0

prints all lines with an even number of fields; and the program

{ $1 = log($1); print }

replaces the first field of each line by its logarithm.

Awk patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, **if-else**, **while**, **for** statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program *grep* [1] will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

{print $3, $2}

prints the third and second columns of a table in that order. The program

$2 ~ /A|B|C/

prints all input lines with an A, B, or C in the second field. The program

† UNIX is a trademark of AT&T Bell Laboratories.

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

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

awk program [files]

executes the *awk* commands in the string **program** on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file **pfile**, and executed by the command

awk −f pfile [files]

1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern    { action }
pattern    { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

1.3. Records and Fields

Awk input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named **NR**.

Each input record is considered to be divided into "fields." Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as described below. Fields are referred to as **$1, $2,** and so forth, where **$1** is the first field, and **$0** is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named **NF**.

The variables **FS** and **RS** refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument **−F***c* may also be used to set **FS** to the character *c* .

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable **FILENAME** contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command **print**. The *awk* program

{ print }

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

print $2, $1

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

print $1 $2

runs the first and second fields together.

The predefined variables **NF** and **NR** can be used; for example

{ print NR, NF, $0 }

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

{ print $1 >"foo1"; print $2 >"foo2" }

writes the first field, **$1**, on the file **foo1**, and the second field on file **foo2**. The **>>** notation can also be used:

print $1 >>"foo"

appends the output to the file **foo**. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

print $1 >$2

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

print | "mail bwk"

mails the output to **bwk**.

The variables **OFS** and **ORS** may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the print statement.

Awk also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in **format** and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints **$1** as a floating point number 8 digits wide, with two after the decimal point, and **$2** as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of **printf** is identical to that used with C.[2]

2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN    { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

```
END  { print NR }
```

If **BEGIN** is present, it must be the first pattern; **END** must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the UNIX text editor *ed*[1] and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, **+** for "one or more", and **?** for "zero or one", all as in *lex*. Character classes may be abbreviated: **[a−zA−Z0−9]** is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and **!**~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly **[jJ]ohn**, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign **$** refers to the end.

2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators **<**, **<=**, **==**, **!=**, **>=**, and **>**. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an **s**, **t**, **u**, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), **&&** (and), and **!** (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with "s", but is not "smith". **&&** and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2 { ... }
```

In this case, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive). For example,

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

prints all lines between **start** and **stop**, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions **sqrt**, **log**, **exp**, and **int**, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s, m, n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s**. The function **index(s1, s2)** returns the position where the string **s2** occurs in **s1**, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1**, **e2**, etc., in the **printf** format specified by **f**. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2**.

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to **x**. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by

```
	{ s1 += $1; s2 += $2 }
END { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are **+**, **−**, **∗**, **/**, and **%** (mod). The C increment **++** and decrement **−−** operators are also available, and so are the assignment operators **+=**, **−=**, **∗=**, **/=**, and **%=**. These operators may all be used in expressions.

3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
	$3 = "too big"
  print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

n = split(s, array, sep)

splits the the string **s** into **array[1]**, ..., **array[n]**. The number of elements found is returned. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

length($1 $2 $3)

returns the length of the first three fields. Or in a **print** statement,

print $1 " is " $2

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

x[NR] = $0

assigns the current input record to the **NR**-th element of the array **x**. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
        { x[NR] = $0 }
END { ... program ... }
```

The first action merely records each input line in the array **x**.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like **apple**, **orange**, etc. Then the program

```
/apple/     { x["apple"]++ }
/orange/    { x["orange"]++ }
END         { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
        print $i
        ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
        print $i
```

does the same job as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

```
for (i in array)
        statement
```

does *statement* with **i** set in turn to each element of **array**. The elements are accessed in an apparently random order. Chaos will ensue if **i** is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while** or **for** can include relational operators like **<**, **<=**, **>**, **>=**, **==** ("is equal to"), and **!=** ("not equal to"); regular expression matches with the match operators ~ and **!**~; the logical operators ||, **&&**, and **!**; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character **#** and end with the end of the line, as in

print x, y # this is a comment

4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed*[1] provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex[3] provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called "report generation" — processing an input to extract counts, sums, subtotals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*;[4] the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".
3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number : ".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls −l*; each line has the form

−rw−rw−rw− 1 ava 123 Oct 15 17:05 xxx

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975. Sixth Edition
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975 .].
4. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975 .].

	Task							
Program	1	2	3	4	5	6	7	8
wc	8.6							
grep	11.7	13.1						
egrep	6.2	11.5	11.6					
fgrep	7.7	13.8	16.1					
sed	10.2	11.6	15.8	29.0	30.5	16.1		
lex	65.1	150.1	144.2	67.7	70.3	104.0	81.7	92.8
awk	15.0	25.6	29.9	33.3	38.9	46.4	71.4	31.1

Table I. Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1.
```
END {print NR}
```

2.
```
/doug/
```

3.
```
/ken|doug|dmr/
```

4.
```
{print $3}
```

5.
```
{print $3, $2}
```

6.
```
/ken/ {print >"jken"}
/doug/	{print >"jdoug"}
/dmr/{print >"jdmr"}
```

7.
```
{print NR ": " $0}
```

8.
```
	{sum = sum + $4}
END {print sum}
```

SED:

1.
```
$=
```

2.
```
/doug/p
```

3.
```
/doug/p
/doug/d
/ken/p
/ken/d
/dmr/p
/dmr/d
```

4.
```
/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\) .*/s//\1/p
```

5.
```
/[^ ]* [ ]*\([^ ]*\) [ ]*\([^ ]*\) .*/s//\2 \1/p
```

6.
```
/ken/w jken
/doug/w jdoug
/dmr/w jdmr
```

LEX:

1.
```
%{
int i;
%}
%%
\n	i++;
.	;
%%
yywrap() {
	printf("%d\n", i);
}
```

2.
```
%%
^.*doug.*$ printf("%s\n", yytext);
.	;
\n	;
```